Numéro d'ordre : XXX

THÈSE

présentée en vue d'obtenir le grade de

Docteur

Discipline: Informatique

par

Pablo Tesone

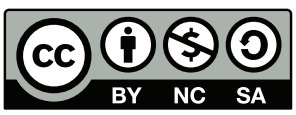
Doctorat de l'Université de Lille
Délivré par IMT Lille Douai

Dynamic Software Update for Production and Live Programming Environments

Soutenue le 17 décembre 2018 devant le jury d'examen :

Président :	Elisa GONZALEZ	Professeur – Vrije Universiteit Brussel (VUB)
Rapporteurs :	Christophe DONY	Professeur – LIRMM – Université de Montpellier
	Oscar NIERSTRASZ	Professeur – University of Bern
Examinatrice :	Elisa GONZALEZ BOIX	Professeur – Vrije Universiteit Brussel (VUB)
Directeurs de thèse :	Stéphane DUCASSE	Directeur de Recherche – INRIA Lille
	Luc FABRESSE	Professeur – IMT Lille Douai
Co-Encadrants :	Guillermo POLITO	Docteur, Ingénieur de Recherche – CNRS
Invités :	Fabien DAGNAT	Professeur – IMT Atlantique
	Noury BOURAQADI	Professeur – IMT Lille Douai

Ecole Doctorale SPI 072 (Lille I, Lille III, Artois, ULCO, UVHC, Centrale Lille, IMT Lille Douai)



Abstract

Updating applications during their execution is used both in production to minimize application downtime and in integrated development environments to provide live programming support. Nevertheless, these two scenarios present different challenges making Dynamic Software Update (DSU) solutions to be specifically designed for only one of these use cases. For example, DSUs for live programming typically do not implement safe point detection or instance migration, while production DSUs require manual generation of patches and lack IDE integration. These solutions also have a limited ability to update themselves or the language core libraries and some of them present execution penalties outside the update window.

In this PhD, we propose a unified DSU named *gDSU* for both live programming and production environments. *gDSU* provides safe update point detection using call stack manipulation and a reusable instance migration mechanism to minimize manual intervention in patch generation. It also supports updating the core language libraries as well as the update mechanism itself thanks to its incremental copy of the modified objects and its atomic commit operation.

gDSU does not affect the global performance of the application and it presents only a run-time penalty during the update window. For example, *gDSU* is able to apply an update impacting 100,000 instances in 1 second making the application not responsive for only 250 milliseconds. The rest of the time the application runs normally while *gDSU* is looking for a safe update point during which modified elements will be copied.

We also present extensions of *gDSU* to support transactional live programming and atomic automatic refactorings which increase the usability of live programming environments.

Keywords: dynamic software update, live programming, long running applications, transactional modifications, automatic refactorings.

Résumé

Mettre à jour des applications durant leur exécution est utilisé aussi bien en production pour réduire les temps d'arrêt des applications que dans des environnements de développement interactifs (IDE pour *live programming*). Toutefois, ces deux scénarios présentent des défis différents qui font que les solutions de mise à jour dynamique (DSU pour *Dynamic Software Updating*) existantes sont souvent spécifiques à l'un des deux. Par exemple, les DSUs pour la programmation interactive ne supportent généralement pas la détection automatique de points sûrs de mise à jour ni la migration d'instances, alors que les DSUs pour la production nécessitent une génération manuelle de l'ensemble des modifications et manquent d'intégration avec l'IDE. Les solutions existantes ont également une capacité limitée à se mettre à jour elles-mêmes ou à mettre à jour les bibliothèques de base du langage; et certaines d'entre elles introduisent même une dégradation des performances d'exécution en dehors du processus de mise à jour.

Dans cette thèse, nous proposons un DSU (nommé *gDSU*) unifié qui fonctionne à la fois pour la programmation interactive et les environnements de production. *gDSU* permet la détection automatique des points sûrs de mise à jour en analysant et manipulant la pile d'exécution, et offre un mécanisme réutilisable de migration d'instances afin de minimiser les interventions manuelles lors de l'application d'une migration. *gDSU* supporte également la mise à jour des bibliothèques du noyau du langage et du mécanisme de mise à jour lui-même. Ceci est réalisé par une copie incrémentale des objets à modifier et une application atomique de ces modifications.

gDSU n'affecte pas les performances globales de l'application et ne présente qu'une pénalité d'exécution lors processus de mise à jour. Par exemple, *gDSU* est capable d'appliquer une mise à jour sur 100 000 instances en 1 seconde. Durant cette seconde, l'application ne répond pas pendant 250 milli-secondes seulement. Le reste du temps, l'application s'exécute normalement pendant que *gDSU* recherche un point sûr de mise à jour qui consiste alors uniquement à copier les éléments modifiés.

Nous présentons également deux extensions de *gDSU* permettant un meilleur support du développement interactif dans les IDEs : la programmation interactive transactionnelle et l'application atomique de reusinages (*refactorings*).

Mots clés: mise à jour dynamique, programmation interactive, applications de longue durée, modifications transactionnelles, réusinage de code

Contents

List of Figures

List of Tables

Introduction

Chapter 1

Contents

1.1 Context

Software needs to evolve to keep up with the requirements of real-world applications, otherwise it becomes obsolete [LB85, DDN02]. During software lifetime, most of the effort is spent during the maintenance phase which consists in adapting existing software to new requirements [DRG+05, KLT03]. Examples of such evolution are: adding new features, improving performance or fixing bugs and security failures.

In the following sections, we present different tools used during the development and maintenance phase of an application. These tools improve the productivity of the developers and minimize the effort spent in those phases.

1.1.1 Live Programming Environments

Live programming environments [San78] such as Lisp [Ste90], Smalltalk [GR83] or Javascript [Dav06], allow developers to modify the code while the program is running. Live programming allows a faster development cycle if we compare it with the *edit-compile-debug* process. Live programming provides a continuous flow of interaction between the developer and the program [BFdH+13, Han03]. This continuous flow of interaction provides an excellent framework for the development of behaviour driven applications [BFL+14, CF17, Lim14, AS14].

Live instances represent the state of an application in an object-oriented programming language. Live Programming environments allow the manipulation of running program's state, through the manipulation of live instances [CNSG15].

Existing live programming tools allow hot update of running code, modifying the structure of live instances as classes change. During code modification, the program is still running. The user of the running application is the programmer itself or other users (*e.g.*, a web application is still serving

content during a live programming session). The programmer modifies the application, debug it and re-execute different parts of the application without the need of restarting the whole application.

The use of live programming is not limited to Smalltalk or Lisp environments. Nowadays, there are different efforts to integrate live programming features in professional programming environments and languages such as Java [WWS10, Zer12, PKC+13], Python [MDB15] and Javascript [nod, fir, chr, ORH02]. This new attempt to integrate live programming in professional IDEs demonstrates the increased interest in the benefits of live programming [BFdH+13, McD07, Tan90, Shn83].

Live programming environments have a limited support to migrate a running application from one version to another. Current solutions are fragile because they mainly consist in applying modifications one at the time and in a non-atomic way. This is even more fragile for the stability and coherence of the running application when applying interdependent modifications sequentially.

1.1.2 DSU Solutions

A Dynamic Software Update (DSU) [HN05, PH13] engine is a tool that manages the migration of a piece of software from version 1 to version 2 while it is running. Its basic idea is to turn the *stop, install, restart* cycle into a simple *update* action [PDF+15]. DSU engines perform such migrations minimizing downtime and guaranteeing that the software will continue working as expected.

DSU solutions (from now on DSUs) are typically used in two scenarios: a production DSU is designed to update long running applications *e.g.,* Web application servers; a development DSU is integrated within a development environment to provide live programming support. Each of these scenarios presents different requirements, making existing solutions to be specialized for only one of them. For example, a production DSU requires safer guarantees while a development DSU requires incremental updates and IDE integration. Not having a single DSU tool limits the benefits of using a DSU solution. The developers need to use different tools with different requirements in both scenarios. The expertise of using a tool in development scenarios is not applicable when updating the application in production.

Current DSU solutions are too specific to either the production or the development scenario. There is a lack of general-purpose DSU. This lack limits the usability of the tools.

1.1.3 Automatic Refactorings

Refactorings are behaviour preserving operations that help developers to improve the design of the application [Fow99, RBJ97, DHL96]. Refactorings modify the implementation of the application keeping its features. This modification of the implementation improves the quality of the code [RBJ97, Rob99].

Nowadays, refactoring tools are present in the majority of Integrated Development Environments (IDE) used in the industry [MT04], but with different degrees of refactoring support.

A refactoring is composed of pre and post-conditions as well as a number of ordered elementary steps. Each step modifies classes and methods. These modifications are performed automatically by the IDE. Automatic refactorings constitute a daily tool used by programmers to improve the quality of its code [KZN12, XS06, DJ05, KZN14, BDLDP+15].

Automatic refactoring tools do not guarantee that live instances' state is preserved after applying the changes.

1.1.4 Reflective Languages

A *Reflective language* has the ability to reason about itself and react upon it [Mae87]. There exists two different forms of reflective access: structural and behavioural [MDC92, MJD96]. The first form inquires on the static structure of a program, and the later inquires on the dynamic running of a program.

Also, the reflective abilities of a system are categorized if the system is able to modify the program structure or its runtime. *Introspection* is the ability to access to the reflective information and *intercession* is the ability to modify the program.

Different degrees of these abilities are required to update running application while it is running. The reflective capacities of a system constraints the ability of update solutions to update a given application running in such systems.

Reflective systems present a Meta-Object Protocol. A Meta-Object Protocol is the set of messages and operations to access and modify reflective systems [KdRB91]. This protocol presents a set of first class citizen object to access the structure of the program and its execution.

A reflective language is structured in two levels:

- The first level includes all the objects and classes that form part of the running application (they describe the business logic that is executing).

- The second level includes a set of objects and classes that describes the running application (they describe the classes in the first level), these objects are the so called meta-objects and meta-classes.

Both set of objects are accessible and manipulable during the execution of the program. Reflective languages present a common environment for objects, classes, meta-objects and meta-classes. An environment is the set of objects, classes and references that are accessible and able to interact with each other. Also, a reflective language presents reification of the concepts used in a program, such as Classes, Methods and Call Stack entries [Riv96a].

Reflective languages are limited in the amount of changes they are able to apply to themselves. They are not designed to apply the changes in an atomic fashion limiting the level of updatability to the core languages and reflective tools [PDFB15, PDF+15].

1.1.5 DSU Scenarios and their Challenges

Updating an application without the need to restart it should guarantee that all the changes are performed and the state of the running application is not altered.

In the scope of an object-oriented application, the state of the application is formed by the live instances in the environment. Guaranteeing the application state in OOP means that live instances should be preserved from one version to the other.

Dynamically updating an application is performed in two different scenarios: Live programming during Development (Development DSUs) and updating a long running application (Production DSU).

Both kinds of DSUs share the challenges listed below. However, each of the families of DSUs present different approaches to solve these challenges. The different approaches focus on improving the results in one of the target scenarios.

- **State Migration.** Migrating the state of an application between versions is not a trivial activity. On the one hand, it requires a technique to replace old values by new values (*e.g.*, pointer swapping, lazy proxies). On the other hand, it requires a way to express value transformations which are usually application dependent and cannot be produced automatically. The migration of state is required to minimize the corruption of instances. A live instance is corrupted if after a change in its structure or its usage it is not updated to follow the required new structure or usage [TPF+16].

- **Change Identification.** Determining the set of changes to apply (*e.g.,* classes to create, methods to modify, instances to migrate) is error prone if done manually. Moreover, doing it automatically lacks precision: the process may miss business dependent value transformations required for state migration.
- **Core Libraries and Self Update.** Updating the core parts of a run-time environment (*e.g.,* core language libraries) and the DSU itself is difficult [PDF+15]. Such updates introduce circular dependencies that may break the update and require special mechanisms to ensure atomicity.
- **Safe Point Detection.** Detecting and deciding the best moment to execute an update (*Quiescence Point*, *Safe Update Point* or *Alterability Point*) [NH09] presents a challenge for those applications that were not designed for it. Looking for a safe update point should be fast enough to minimize the suspension time and smart enough to detect as soon as possible when such update point will never happen.
- **Execution Penalty.** Implementing all the above requires smart strategies to avoid performance penalties outside the update itself. For example, the usage of lazy proxies for state migration introduces an additional level of indirection affecting the overall application performance.

Table 1.1 summarizes how the different kinds of DSUs choose to balance their features to cope with these challenges.

Challenge	**Development DSUs**	**Production DSUs**
State Migration	Limited	Yes
Change Identification	Automatic	Manual
Core lib & Self Update	Limited	No
Safe Point Detection	No	Manual
Execution Penalties	Most	Most

Table 1.1: How different DSU solutions face the challenges.

1.2 Problem Statement

This thesis focuses on the development of techniques to perform dynamic software update in Live object-oriented Reflective Environments that are applicable in both development and production scenarios.

In this context, we aim solving the following problems:

State Migration. Migrating live instances from one version of the application to a different one requires the migration of instances. This operation requires the identification of the changes that requires instance migration, the automatic migration of the instances when it is possible and the tools to provide to ease the manual migration.

Safe Point Detection. To safely update an application the modifications should be done in a point in time that does not affect its normal execution. This thesis focuses on a conservative approach to guarantee a safe point to perform the update.

Isolation. While the developer is performing Live programming the changes she is performing should not affect the running application until all of them are executed.

Execution Penalty. Using a DSU solution should not affect the execution performance of the application. Moreover, the only impact of the use of a DSU solution should be noticeable during an update window.

Usage in Production and Development scenarios. To ease the use of DSUs in the daily development cycle, an approach should be able to be used in both scenarios.

Development Environment Integration. To ease the use of DSUs in a daily development cycle, an approach should be integrated with the IDE used by the developer. It should collaborate with existing tools in the development environment as Automatic Refactorings.

Then, we pose the following research question:

> *What would be general-purpose DSU solution, integrated with the development environment of object-oriented reflective languages, to be used both in production and development scenarios, and guarantee the correct state migration, safe point detection and isolation without affecting the execution performance outside the update window?*

1.3 Contributions

The main contribution of this thesis is *gDSU* and the techniques required to implement it. *gDSU* is a DSU solution that is applicable in production and development scenarios. It is applicable to object-oriented Reflective Languages and provides a safe live update experience and update of long running applications.

The proposed solution provides a set of automatic and manual migration strategies for instance migration. It implements a conservative safe point detection algorithm. It guarantees the isolation of changes and it does not present any execution penalty outside of the update window. Also it is integrated in the IDE providing transactional change support and handling of complex automatic refactorings.

1.4 Thesis Outline

The rest of this dissertation is structured in three main parts followed by conclusion chapter and several appendixes with complementary information.

1.4.1 Part I: State of the Art

Chapter 2. This chapter presents the state of the art of dynamic software update solutions and related techniques. The first section of this chapter presents an analysis of the requirements for a general-purpose DSU solution, and how these requirements are related with the development and production scenarios. Then, this chapter presents a representative set of DSU solutions and classify them in a taxonomy based on the intended usage scenario. This chapter also describes and analyses some the related techniques used in DSU solutions. Finally, the chapter presents an analysis of the categories of solutions using the stated requirements.

1.4.2 Part II: DSU for Production

Chapter 3. This chapter presents the design principles of our proposed DSU solution. A general description of our solution is presented, later each of the mechanisms and techniques used in the solution is explained. The presented solutions address the different requirements for a general DSU solution.

Chapter 4. This chapter presents the design of the techniques used to implement an efficient *gDSU*. In this chapter, efficient techniques are presented to implement the functionality needed by *gDSU*. All these techniques allow us to develop an efficient DSU solution that minimizes the performance issues and ease its usage [TPF$^+$eda].

Chapter 5. This chapter presents the validation of the production DSU aspect of *gDSU*. First, it presents three different validations to demonstrate that *gDSU* is able to do an application update, update itself and the core language libraries. Moreover, this chapter presents a benchmark showing the minimal memory footprint and execution penalty of *gDSU* [TPF$^+$eda]. Finally, this

chapter presents an assessment of the requirements established for a Production DSU.

1.4.3 Part III: DSU for Live Programming

Chapter 6. This chapter starts covering the requirements for a live programming DSU. Initially this chapter analyses the challenges of executing automatic refactorings in a live programming environment. Based on the set of problems shown in this chapter, an extension for *gDSU* is presented to support atomic automatic refactorings. Finally, this chapter presents the implementation considerations of this extension [TPF+edb].

Chapter 7. Following with addressing of the live programming problems, this chapter analyses the need for a state-aware transactional live programming. This chapter proposes another extension to *gDSU* that allows us to support state-aware transactional changes. Later, this chapter presents the techniques that have been required to implement the proposed extension [TPF+18b]. Finally, this chapter presents a requirement assessment of the challenges for a live programming DSU.

1.4.4 Part V: Conclusion

Chapter 9. This chapter concludes this dissertation. It sums up the contributions of this thesis and presents several lines of future work.

Appendix A. This appendix presents a list of the publications that came out of the work on this thesis.

Appendix B. This appendix presents detailed instructions to reproduce the validations and benchmarks presented in this dissertation.

Appendix C. This appendix presents the detailed analysis of the automatic refactorings and the problems they present when used in a stateful live programming environment.

Part I

State of the Art

Comparing Existing Solutions

Chapter 2

Contents

This chapter first draws a list of requirements for a complete DSU engine. It then describes and classifies main existing DSU solutions and related techniques. Using this classification, it finally presents a comparison of families of DSUs solutions based on the previously established list of requirements.

2.1 Requirements for DSU

This section starts by presenting the challenges that arise when applying changes to a running application through an example. These challenges are then used to establish a list of requirements for a DSU solution to tackle both production and development scenarios.

2.1.1 Change Challenges Illustrated

Let us consider an application running a 3D visualization on a continuous stream of data. This application has a Window class that implements a drawOn: method responsible for rendering a single frame of our visualization in a 3D canvas. This application has a render thread with a loop invoking the drawOn: method. So, Window continuously draws in its canvas instances of Vector3D class. Figure 2.1 shows a screenshot of such application and Figure 2.2 illustrates its code.

Let us consider now that a developer wants to change the coordinate system from cartesian coordinates to spherical coordinates for performance reasons. Figure 2.3 presents the desired modification. This modification requires a number of different changes:

- The Window»drawOn: method will be updated to use the new coordinates of Vector3D.

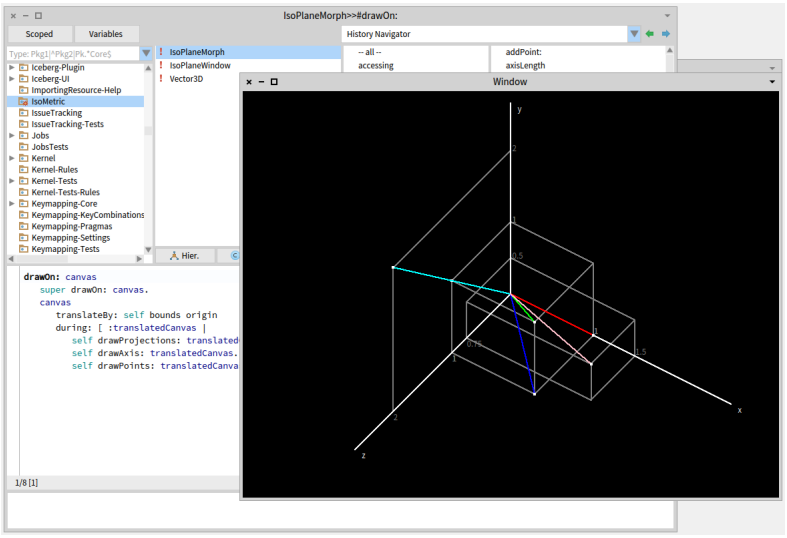


Figure 2.1: Window of the rendering application in action.

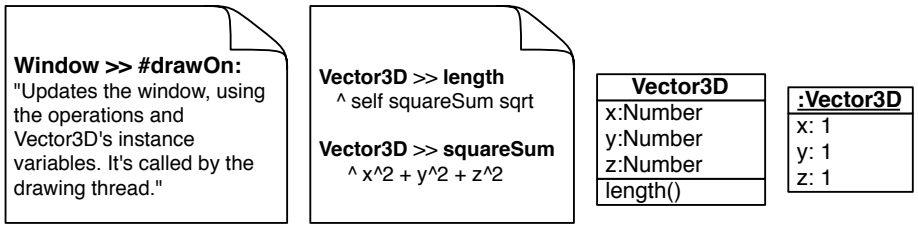


Figure 2.2: Original version using cartesian coordinates

- The Vector3D»length method will be replaced to use the *radius* instance variable.
- The Vector3D»squareSum method will be removed, as it is not used anymore.
- The structure of the class Vector3D will be changed replacing the existing instance variables with the new ones.

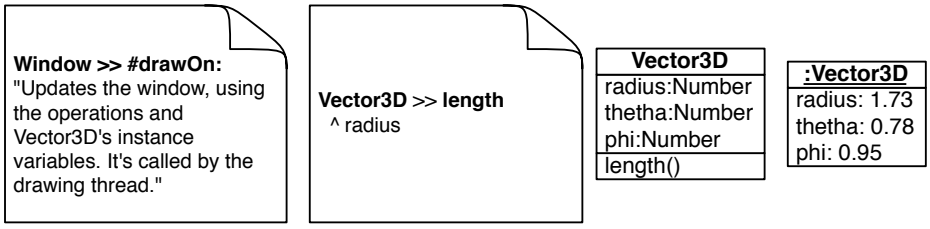


Figure 2.3: New version using spherical coordinates

Applying these changes while the application is running is not a trivial task. This example clearly shows the three problems that occur when updat-

ing a running application:

State Inconsistency. Updating an application involves changing its internal state representation. However, naively initializing such state may produce information loss or even failures in the application. Most of the times, a migration is business dependent. In our example, the Vector3D instances change: initializing the new instance variables to null produces a null pointer exception, while initializing them to 0 will produce a loss of previous application state. Instances should be migrated from their cartesian coordinates to their corresponding spherical coordinates. The logic of this migration is clearly business dependent and cannot be generally inferred just from the changes in the classes and methods (Section 2.1.3).

Change Interdependency. When updating an application, the modifications are usually interdependent because they relate to the same entities (*i.e.,* methods, instance variables, classes). In the example, the new Vector3D»length implementation requires the previous introduction of the new radius instance variable. Modifying this method before adding the instance variable is wrong, and it causes application failures. These interdependencies appear also between methods. For example, removing Vector3D»squareSum before updating Vector3D»length produces a missing method error during the execution of Vector3D»length (Section 2.1.4).

Concurrency and Execution Inconsistency. While the application is running, some of the methods to be updated are present in the execution stacks of running threads. The update process should not lose or corrupt the application execution. Such a corruption occurs when the update alters local variables or control-flow of a method in any execution stack [HN12]. In our example, if the render thread enters the Vector3D»length method and the update is applied before the Vector3D»squareSum method is called, continuing the execution of the length method will fail as the Vector3D » squareSum method does not exist anymore (Section 2.1.5).

These issues make applying code changes a challenging and interesting task, to which existing DSUs propose several techniques to solve. However, implementing a solution in a way that it is applicable in a real scenario imposes new challenges. The following section explores those challenges concerned with these more practical issues.

2.1.2 DSU Practical Concerns

DSUs usage presents a set of concerns that should be addressed to have a practical solution.

Performance. Making a program updateable should impact its performance as little as possible [HN05]. A DSU performance impact is divided in two stages: (1) during normal execution (outside update-window) and (2) during an update (inside update-window). A DSU should minimize the impact in both stages. Examples of impact are memory consumption, execution overhead and downtime during the update (Section 2.1.6).

Ease of Use. The DSU engine should be easy to use by regular application developers. The less complicated the updating process is, the less error-prone it will tend to be [HN05]. A DSU solution should be integrated with the development tools used in the language. Also it should minimize manual interactions and simplify them when they are unavoidable. Finally, it should be present in the whole life-cycle of the application providing solutions during the development as well as during the evolution of systems in production (Section 2.1.7).

Versatility. A DSU should be able to update any part of the running application [HN05]. The running application is not the only part that may require modifications. Core language libraries including the DSU engine itself also require updates (*e.g.,* adding new features, improving performance or fixing bugs and security failures) (Section 2.1.8).

2.1.3 State Inconsistency

Applying updates to live programming environments change the structure and use of live instances. There are changes that leads to instance state inconsistency. Instance state inconsistency is when the internal state of live instances does not follow the expected value for live instances. This mismatch includes differences in existence or not (instance variables that exists in only one of the version), how the instance variable is used (the same instance variable is used for different operations in different versions) and values types (the instance variable holds different types of values in different versions). These instances should be correctly migrated to avoid instance state inconsistency.

A DSU in an object-oriented environment should manage arbitrary combinations of the following elementary changes. These changes are composed in different ways, allowing the creation of complex changes. Including

changes in many classes at the same time, for example complex automatic refactorings.

Adding new instance variable. Adding a new instance variable on an existing instance means extending the structure of the object and filling the room with a value. Neither initializing the new variable with the null pointer nor using the value assigned in the construction are useful for existent objects. The decision of the value for the new instance variable depends on the update performed.

Removing instance variable. When removing an instance variable there are two options. Either the object is resized to fit its new layout and the value is dropped or the object keeps the value but the instance variable is made obsolete as it is not a property of its class any more.

Renaming instance variable. From the structure point of view this is equivalent to removing an instance variable and adding another one. But doing so, we loose the value held in the old instance variable. Therefore, the system needs to be able to transfer state from an instance variable about to be removed to a newly added instance variable.

Value Change. When the object structure is updated, it might not be a structural change. Changes are application specific and only affect the value stored in an instance variable without modifying the structure of the object. These migrations may need to compute the new state of the object in various manners that involve all the data available in the object. The way the new values are calculated are application dependent. For example when in an object representing a product we store a price as a number. In a later version we store the price as an object (containing the number and the currency). The structure of the instances are not modified (both versions have the *price* variable), but the instances should be migrated and the migration is application-dependent *i.e.*, what is the default currency.

Class Hierarchy Changes. A class hierarchy change can generate or not changes in the structure of the instances. The DSU should detect when these changes affect the structure of the instances and migrate the affected instances.

2.1.4 Change Interdependency

Changes in an update are dependent between each other. This dependency is produced when the same element (*e.g.*, classes, methods) is modified in

two different changes, and when different elements have changes that are required by the other.

An example of the first scenario is a class that is modified to have a new instance variable and a method is added to this class using such instance variable. Figure 2.3 presents an example of this change. The method Vector3D » #length depends in the modification of the structure of the class Vector3D.

An example for the second scenario is a a method that is modified to use a new method in other class. The method Window » #drawOn: has a dependency with the implementation of Vector3D » #length

This interdependency of changes requires that the changes are applied in correct sequence. The dependencies should be correctly managed during the update by the DSU solution.

A correct update includes all the changes applied in the system. These dependency also affects the correct migration of live instances.

2.1.5 Concurrency And Execution Inconsistency

During the update the application is execution, so it is possible that in the execution stack there are activations of methods that need to be updated. Also, the execution stack could include references to objects that should be migrated.

The execution state of the application should be guarantee during and after the update. All the elements in the execution stack should be correctly updated or migrated.

The DSU should execute the update only when it can guarantee that the execution stack of the application is not affected. Also, in multithreading applications the updates and migrations should be synchronized.

From the point of the running application the update should be performed transparently and atomically.

2.1.6 Performance

Executing an update impact in the execution performance of the running application. A DSU should minimize the impact on the running application.

A DSU impact the performance of the running application if the application has an execution penalty because of the execution or presence of a DSU solution.

There exist two different situations during the update window, and outside the update window. On one hand, It is required that the DSUs do not affect the execution of the application outside the update window. On the other hand, the DSUs should minimize the length of the update window as the application is not executing during this window of time.

2.1.7 Ease of Use

If the DSU solution requires manual work from the developer, this should be minimized.

Operation as the identification of changes, creation of a patch, expression of state migration logic and detection of safe point update should be performed automatically.

In the scenario where the operations could not be completely automatic, the requirement of interaction from the developer should be minimal.

Moreover, a DSU solution for live programming in development environments should give the ability to perform the changes in a iterative way. The developer should be able to perform the changes incrementally testing each of the modifications committing or discarding them without affecting the stability of the running application.

Finally the DSU should present a common usage interface and configuration for changes made in development and production. This transparency for the user improves the usability of the tool and the use during the whole development process.

2.1.8 Versatility

A DSU should be able to update not only the application code, but also core language libraries and the DSU engine itself. This ability is crucial to support unexpected updates, improvements in the application, DSU tool and the core language. This characteristic is useful in both development and production scenarios.

In the first scenario it allows the safe modification and extension of a live programming environment. In reflective languages, modifying the core libraries and tools could lead to instability of the environment as the tools used to develop the application are using the modified code. Having the ability to modify the core libraries allows the developer to perform live programming not only its own application code but also in the whole system. The DSU solution should guarantee the correct isolation of the changes in a live programming environment until the developer decides to apply the whole set of changes.

In production scenarios, long running application could require updates in any of the elements of it, *e.g.*, security updates in the core libraries. It is not possible to know in advance the elements that should be modified in the future.

2.1.9 Requirements for a General DSU

We consider a general DSU as a tool capable of updating applications in development and production scenarios. Taking in consideration the problems and concerns a practical general DSU should overcome, we have enumerated a set of requirements. Table 2.1 presents how the requirements cover the problems and concerns that we explain here after.

Problem	**Concerns**
State Inconsistency	State Migration.
Change Interdependency	Atomicity.
Concurrence and Execution Inconsistency	Automatic Safe Point Detection.
Performance	Small Run-time Penalty. Minimal Application Downtime.
Versatility	Self and Core Lib Update.
Ease of Use	Patch Generation. Patch Reuse. Broad Applicability.

Table 2.1: Requirement mapping to the problems and concerns of a DSU.

Atomicity. A DSU solution should perform atomically all changes in a single update. All the changes should be applied at once, or at least, the execution and state of the old and new versions should not be mixed.

State Migration. A DSU solution should provide the means to migrate the state of the application from one version to the other. The needed migration logic might be produced automatically or provided by the developer. The migration logic for value transformations that depends on business logic cannot be generated automatically thus, a DSU solution should minimize required manual interventions.

Automatic Safe Point Detection. A DSU solution should detect the safe points to perform an update. As the application is running the program under update, the update should be performed while it does not have any impact on the running application or the solution should handle the impact. The detection should be done minimizing developer intervention.

Patch Generation. A DSU should automatically calculate the set of changes needed to pass from one version to the next one. This set of changes is a

patch [SAM13]. A DSU should provide a clean integration with the development process providing *programmer transparency* [MME12]. This integration minimizes the need of manual intervention in patch creation.

Patch Reuse. If the DSU solution requires the participation of a developer, it should allow the developer to reuse and compose these elements. The reusing and composing of migration logic and patches ease the developers' manual work.

Self Update and Core Lib Update. As the bugs and improvements not only occurs in the application under modification, it is required that the DSU solution allows the developer to modify the core libraries of the language and the DSU itself.

Small Run-time Penalty. A DSU solution should minimize the performance impact it has on the application during its normal execution *i.e.,* outside the update window. Ideally, the application should run as if there is no present DSU solution. As a counter-example, techniques such as bytecode manipulation or lazy proxies introduce an additional level of indirection affecting the normal performance of the application.

Minimal Application Downtime. A DSU solution should minimize the downtime of the application during the update window.

Broad Applicability. A DSU should be applicable during the whole lifecycle of the application. It should be applicable in development and updating an application in production.

Isolation. A DSU should isolate the changes until the full set of changes is committed. In a live programming experience, the set of changes is created incrementally and during the development phase these modifications should not affect the running application.

2.2 Existing DSU Solutions

This section describes a series of DSU solutions for object-oriented languages running on top of a Virtual Machine. The selection of DSU tools presented represent different characteristics and they are designed to different use case scenarios. This selection represents examples of different DSU solutions. Other DSU solutions not present in this list use similar strategies and solutions of the ones included in the selection.

2.2.1 DUSC

DUSC [ORH02] is a Java DSU solution based in class swapping. It is performed in two stages. In the first stage the application is rewritten using bytecode manipulation to be able to be updated. Each use of a class is replaced by the use of an interface. The real implementation of the class is used behind proxies. In the second stage, during an update the implementation classes are modified.

It is required that the application code does not directly access to instance variables. It should use accessors. Also, the application code should not use reflective calls. Also global state and static methods use should be limited or they are not able to be updated. It detects a safe update point by checking that there are no modified methods in the call-stack. Figure 2.4 presents the transformation of a class to be update-ready.

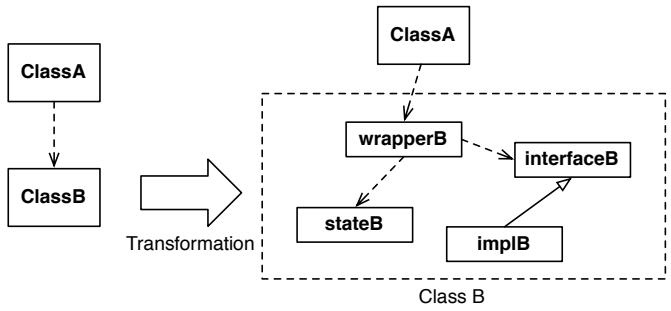


Figure 2.4: Static Transformation performed by DUSC to enable dynamic update of the class.

This solution does not require modifications to the virtual machine, but requires bytecode manipulation while the application is loaded and proxy logic is added to the execution of the application. The update-ready classes are divided in four classes: (1) wrapper, (2) state, (3) interface and (4) implementation. By diving the classes, it is able to update part of the classes without affecting the clients. However, this modification limits the ability to debug the application as the runtime version of the application does not correspond with the development version. As the client code has references to the interfaces of the update-ready classes, the updates cannot modify the interface exposed by a given class. By a limitation of the JVM, the old classes are never discarded. Even though they should not be accessible, old versions of the classes are present in memory. DUSC does not include support for custom state migration.

2.2.2 Jvolve

Jvolve [SHM09] is a DSU solution for updating Java programs. Jvolve implements a virtual machine level DSU, so it needs a special version of the Java Virtual Machine to run. It is implemented using a modifying version of Jikes RVM[1].

Jvolve is able to handle adding, replacing and removing fields and methods. It handles changes in any level of the class hierarchy. To initialize new fields and update existing ones, Jvolve applies class and object transformer functions. Jvolve includes automatic generation of transformers for default values. Complex state migration including changes in the business logic requires the manual development of transformers.

Jvolve uses bytecode verification to statically type-check the updated elements. Also it uses bytecode manipulation to introduce barriers and trampolines to handle the automatic detection of safe update points. It extends the Garbage collection implementation to handle the migration of state.

When the application is outside the update window, the execution penalty is minimized.

2.2.3 DCEVM

DCEVM [WWS10] is a DSU solution for Java. It is implemented as a modification of the Java Virtual Machine. It extends the JVM debugging interface with new operations. These operations perform modifications that are not possible in the stock implementation of HotUpdate (*e.g.,* modification in the hierarchy, modification in the class structure).

DCEVM keeps the different versions of the classes in the system. During garbage collection, it updates the live instances that requires an update.

It does not implement a way of expressing custom state migration from a version to another. This solution does not perform safe update point detection to guarantee the stability of the running application.

2.2.4 DuSTM

DuSTM [PC11] is a Java DSU solution. It is implemented using multiple memory transactions. DuSTM requires bytecode rewriting to replace the uses of application classes and live instances with transactional proxies. DuSTM divides the user classes in different classes in the same way as *DUSC*. It preserves the identity of the objects by using a wrapper as a proxy of the instances. This transactional proxies are used to handle the update of the application. It introduces a lazy upgrade model, where the changes

[1] https://www.jikesrvm.org/

are applied as the code is executed. It also presents a state migration API to express complex state migration. DuSTM is limited in the amount of class hierarchy changes it handles.

2.2.5 JRebel

JRebel [Zer12] is a DSU tool designed to update Java programs. It is implemented to run on top of the stock Java Virtual Machine. It implements state migration and it minimizes runtime and memory penalty.

JRebel instrument the application and libraries to introduce a layer of indirection. This layer of indirection is introduced by bytecode analysis and manipulation. This layer of indirection implements a proxy between the live instances and other representation in memory. Each live instance points to a real instance that corresponds with the current version of the application. To update the application all the proxies are updated to point to instances of the correct version.

JRebel does not support changes in the hierarchy of classes and interfaces. It implements a basic initialization of new instance variables and it does not support changes in instance or global state. Moreover, modifications that require the change in the types of an instance variable are not supported.

These limitations restrict the update possibility in stateful long running applications.

2.2.6 Javeleon

Javeleon [GJK+12] is a DSU tool developed to update Java programs. It leverages the classloading mechanism to have multiples versions of the application running at the same time. It introduces proxies to replace instances with new versions of the updated classes when an updated instance receives a message.

Javeleon requires bytecode instrumentation and class modification during the bootstrap of the application. It does not handle custom state migration. It uses the instance initialization to generate the values for added instance variables.

Javaleon requires correct construction of the patch to update all the clients a given class in the same update. As it uses proxies, the updated application should be correctly built to minimize the inlined methods and direct access to the instance variables. Direct access to instance variables and inlined methods are not accessed through the proxies, so they cannot be updated correctly.

This solution is integrated with the NetBeans IDE. It generates the patchs from the modifications in the IDE.

2.2.7 Javadaptor

JavAdaptor [PKC+13] leverages the HotUpdate support present in the Java Virtual Machine. It replaces the modified classes with new versions of them. It updates all the references to the older classes with new classes.

JavAdaptor is integrated with the Eclipse IDE, and it generates the patch from the modification in the IDE. It has limited support to handle automatic refactorings, although it does not handle the migration of live instances.

JavAdaptor uses proxies and delegation, having an impact in the execution of the application. It uses bytecode manipulation to introduce DSU related logic in loaded classes. It does not support the modification of core libraries limiting the modifications to the application.

JavAdaptor includes some of the characteristics of Production DSUs, although it lacks support to express complex instance structure migration (*i.e.*, renaming variables, changing the type of variables, changing from a native value to an object). Lacking this support is not a problem in development environments, but it limits its usability for production environments.

2.2.8 Rubah

Rubah [PH13] is a DSU solution for Java programs running in the stock Java Virtual Machine. It is implemented as a library and it uses bytecode rewriting to modify the running application to perform the dynamic update. It introduces two schemas of state migration: one lazy using proxies and other eager. It requires tailored modifications of the program to support safe update point detection and minimizing the update window. Regarding patch generation, Rubah implements an API for the update of a program. The developer should express the changes implementing the update logic. This logic is executed and it is responsible of performing all changes. The update objects can extend already implemented solutions.

2.2.9 Pymoult

Pymoult [MDB15] is a Python library implementing various DSU mechanisms. Pymoult is used to evaluate different DSU mechanisms. Using different mechanisms, Pymoult addresses different type of users.

Pymoult requires an special version of the PyPy interpreter to execute. It executes all

Pymoult presents an API for describing the update of the program. The developer should express the changes implementing the update logic. This logic should include all the changes in the update. The update logic is reusable in different updates.

The program should be rewritten to handle the detection of safe update point, thread handling and state migration. It does not handle self update or core libraries update.

The isolation and the atomic application of changes requires of a correct development of the update and the modifications in the target application.

2.3 Categories of Existing Solutions

Existing solutions have been classified in two categories: DSUs designed to be used in production environments (Section 2.3.2) and DSUs designed to be used in development environments (Section 2.3.3).

It is true that many of these solutions are usable in both scenarios. However, as they are designed to satisfy the requirements of one of these scenarios, the other scenario's requirements are not fully satisfied. For example, DSUs designed for production environments do not provide automatic generation of patches limiting its usability as a development tool, but they provide an extensive API for state migration that is not present in DSUs for development environments.

Classical Live Programming environments such as Lisp or Smalltalk (Section 2.3.1) have been considered within the development oriented DSUs. However, in this section we study them separately. Although they provide programming language support for DSU scenarios, they are not fully engineered for this task and considering them amongst development DSUs would be unfair.

2.3.1 Classical Live Programming Environments

Live programming environments [San78], such as Lisp [Ste90], CLOS [KR90], Smalltalk [GR83] and Javascript environments allow developers to modify the code while the program is running. These tools allow hot update of running code. When the class structure changes, the structure of live instances is updated. Often these languages offer a Meta-Object Protocol (MOP) [KdRB91] to support version migration of instances and code modification [Riv96b].

However, the migration support is limited. For example, when new fields are added, these new fields are left uninitialized. In addition, these solutions do not apply the changes atomically. They apply the modifications one at the time. As they do not implement atomicity, the sequencing of changes is mandatory. As sequencing is not always possible [PDF+15] they have limited ability to update core libraries. Also they do not handle the concurrency or the detection of safe update points.

These solutions are designed to be used during development. The required support is implemented in the language infrastructure and they do not produce additional run-time impact. Patch generation is done automatically because the IDE is integrated with the update tools.

Finally, live programming tools for Javascript (*e.g.,* Nodemon[2], Firebug[3], Chrome Dev Tools[4]) do not handle the migration of live instances.

2.3.2 Production DSUs

DSUs designed for production (*e.g.,* Pymoult, Rubah, DuSTM, DUSC) provide the means for applying an update atomically and provide state migration mechanisms. They are designed to minimize the downtime during the execution of the update. While they require the manual generation of the patch, they provide reusable elements to compose the patches. Also the patches allows the expression of limited instance migration logic. They present limited or none IDE integration. They provide limited self and core update.

Production DSUs present run-time impact as they require to rewrite the application to be update-ready. Also, they require bytecode manipulation to introduce proxies. These proxies execute during normal execution of the application. They generate an impact in the size of the program and a penalty in the execution. These solutions present an execution penalty during the normal execution of the application, affecting the application outside the update window.

2.3.3 Development DSUs

Development DSUs such as Javeleon, Jvolve, DCEVM and JRebel are intended to be used during the development of the application. They are integrated with the language IDE and generates the patches automatically. However, they do not allow custom state migrations.

As they are designed to be used during development, these solutions were not designed with performance impact in mind. They require modifications in virtual machines to run. As an exception, JRebel does not require VM modifications but the amount of changes it can handle is limited (*e.g.,* it does not allow hierarchy changes).

In all these solutions, the DSU related code is running all the time impacting the global performance of the application even outside the update window.

[2]https://github.com/remy/nodemon
[3]https://addons.mozilla.org/en-US/firefox/addon/firebug/
[4]https://developer.chrome.com/devtools

Development DSUs are not able to apply changes to the DSU engine or the language core libraries.

During the development process the developer perform a series of complex changes. A known set of complex changes are the automatic refactorings. These complex changes lead to instance corruption. Following we present the state of the art of handling Automatic refactorings in live programming environments and Development DSU solutions.

The Rewrite Engine [RBJO96, RBJ97] provides a complete refactoring tool. Since 1996, it has been used in different Smalltalk implementations like Pharo, Dolphin Smalltalk, VisualWorks and VASmalltalk. However, this tool does not support migration of live instances. Our solution handles the correct migration of live instances.

Development DSU are integrated with the IDE. However, they are not integrated with refactoring tools. The refactoring operations are performed on the source code. Even though having the support to migrate the instances, the developer is in charge of generating the migration patch and not the refactoring tools. The refactoring tools are not aware of the DSU tools, and the IDEs handle the changes statically only. Our solution introduces the integration of both tools. In the related works, the tools are not correctly integrated, depending on the developer to implement the changes performed by the refactoring by a DSU tool.

Moreover, in the work describing JavAdaptor [PKC+13], it have been used to implement automatic refactoring but the migration of instances have not been addressed. The lack of instance migration reduces the applicability of the solution. Our solution provides an integrated solution.

Working with a dynamic language does not change the disconnection between the DSU tools and refactoring tools. An example of this is Pymoult [MDB15], that provides the support to migrate live instances after a refactoring operation. But none of the tools working on a live Python environment uses this support.

The dynamic nature of Javascript allows the change of the running code and modification of the instances. However, the refactoring tools present in Javascript development environments such as WebStorm [Web], Grasp [gra], Atom [ato] and Visual Studio [vis] handle all the changes in source code level, without caring about live programming.

Also the live programming tools for Javascript as Nodemon [nod], Firebug [fir] or Chrome Dev Tools [chr] do not handle the migration of live instances. They only migrate the code generating these instances. The live programming experience in these environments is not comparable, as they are not intended to implement complex automatic refactorings.

As we shown, the infrastructure to integrate DSU tools and automatic

refactorings in IDE exist in the related work, although the transparent integration that we propose is not present in the related work.

2.4 Related Techniques

The literature describes many other techniques that are not a full DSU implementation but could be applied to define a DSU solution. In this section we compare several such solutions.

2.4.1 Safe Point Detection

Regarding safe point detection techniques, Cazzola et al. [CJ16] propose an automatic way to determine unsafe update points. Their technique uses static analysis of the changes to detect unsafe update points. Then the update process uses this information to avoid performing an update in those points. This solution could not handle the case when the problems arise from third party libraries, where the source code is not available.

Makris et al. [MB09] propose a way of updating software, without waiting for a safe update point, but in their solution the application source code should be instrumented before execution.

Buisson et al. [BCD+14] propose a way of formally validating safe point detections. They guarantee the safeness of the update thought the use of a formal model of the application. However, it requires the generation of a formal model of the updated application.

2.4.2 Migration Logic Generation

Magill et al. [MHSM12] propose a way of automatically generate the migration logic for a set of example objects. The user must provide a set of examples in the old and the new version and their solution synthesizes a transformation function for each of the pairs of objects. These functions are later used during the update to migrate the instance state. This solution does not handle the totality of the migration scenarios but provide a baseline for manual modification by the developer. However, it requires the developer to provide a set of tests or executions that generates the heap snapshots for both versions.

Penney et al. [PS87] propose a way of handling modifications, but they require modifications in the virtual machine. Wernli et al. [WLN13] propose to manage isolation using a copy and keeping alive both the old and new environment. The access to the modified objects is done through a lazy proxy, adding an execution penalty to the application after the update. This also produces problems with the identity of objects because the updated instances are duplicated.

2.4.3 Benchmark and Validations

Regarding the safety guarantees provided by the DSU, Tedsuto [PH16] is a tool that provides a general framework for testing updates and detecting if they are successful. However, it requires manual intervention to create the validations and the invalid environment should be discarded so it cannot be applied to a productive environment.

2.4.4 Architectural Solutions

There are also architectural alternatives to DSU [KM85,OMT98,PBJ98]. However, these solutions should manually take care of replication and persistence of application state. They require more complex and manual update schemes and special handling of running instances and processes. Also, they are only designed to handle anticipated update situations. Because of this, this thesis left out of the scope these solutions.

2.4.5 Isolation and Atomicity

Mattis et al [MRH17] implement transactional support for Squeak, they allow to modify the classes and methods scoping the changes to a set of threads. They also support to apply the changes back into the original environment. However, they do not propose any support to state migration or conflict detection.

Denker et al [DGL+07] propose a transactional solution *Changeboxes*. Changeboxes provide an object model representation of changes and scope the changes in a thread-local context. However, changeboxes are not intended to apply the changes back into the original environment, but to co-exist with the original environment. Also, they do not handle the migration of state or the detection of possible conflicts.

Lincke et al [LH12] propose a way of scoping the changes in a live programming environment. However, they do not allow to apply back the changes to the original environment and their solution does not arise all the problems of a class-based system, as it is based in a prototype system.

Wernli et al [WLN13] propose a way of scoping changes inside contexts. A context scope the changes to the classes and methods, and the logic to migrate from one version to another and back. Their solution is designed to allow different versions of the same application to co-exists. They provide lazy migration from one version to the other. However, they do not implement detection of conflicts and requires more migration logic to interact with all the co-existing versions of the application. Our solution only keeps a single version of the running application, the alternative environments are only to

edit and test code. There is no need to keep the different version running at the same time.

Casaccio et al [CPDD09] propose a technique for editing Pharo images without affecting the running environment. They propose to edit a client image from a server image. The modifications in the client image does not affect the server image. However, this solution requires to copy the whole image and does not include a way of applying back the changes in the running application.

2.5 Analysis of Existing Solutions

This section compares existing DSU approaches. To compare these solutions we use the requirements stated before. Table 2.2 presents the results of the comparison using the proposed requirements.

Category	**Requirements**										**Examples**
	Atomicity	State Migration	Automatic Safe Point Detection	Small Run-time Penalty	Minimal Downtime	Broad Applicability	Isolation	Patch Generation	Patch Reuse	Self and Core Update	
Classical Live Programming	○	◐	○	●	●	○	○	A	○	◐	Lisp, Clos, Smalltalk
Development DSUs	●	○	○	○	◐	○	◐	A	○	○	Jrebel, Javeleon, Jvolve
Production DSUs	●	●	◐	○	●	○	●	M	●	◐	Rubah, DuSTM, Pymoult

●: Yes ○: No ◐: Limited
A: Automatic M: Manual S: Semi-Automatic

Table 2.2: Requirement vs. DSU Categories

Part II

DSU for Production

Design Principles of gDSU

Chapter 3

Contents

This chapter presents a new DSU solution called General Dynamic Software Update (*gDSU*). *gDSU* is called general because it is applicable to both productive and development scenarios fulfilling all their requirements in a practical way. *gDSU* allows developers to apply changes to an application, core language libraries and even on the DSU engine itself.

3.1 *gDSU* in a Nutshell

The entry point of the *gDSU* engine is a patch. A patch is a collection of changes that describe the update to perform. It includes the changes to apply in the code, such as method and class modifications, and the corresponding state migration logic.

gDSU generates this patch semi-automatically by applying a version control system (VCS) version diff or/and by recording code changes made in the IDE during a development session. In both approaches, business-related state migration policies should be provided by the developer (Sections 3.2 and 3.3).

When *gDSU* is used in production, the developer generates a patch in her development environment. Afterwards, she applies the patch using the *gDSU* engine deployed in the production environment. In the case of a live development scenario, the patch is generated and applied at the same moment in the development environment.

gDSU takes the patch as input and performs the following steps to safely apply the update. These steps are the same regardless the update scenario. Figure 3.1 illustrates such steps.

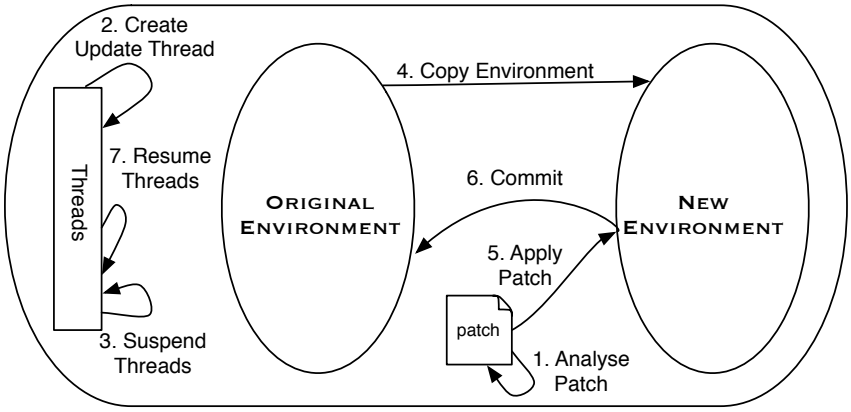


Figure 3.1: Steps to apply an atomic update

1. Analyse Patch. *gDSU* analyses the patch and calculates the changes to perform in the environment. This analysis is used to detect what live instances it should migrate and the conditions to reach a safe update point (Sections 3.4).

2. Create Update Thread. *gDSU* spawns *the update thread*: This thread is responsible of monitoring the other running threads looking for a safe update point and performing the update (Section 3.5).

3. Suspend Threads. When the safe point is reached, the update thread suspends all the other threads and the update process can begin (Section 3.5).

4. Copy Environment. *gDSU* copies the instances and classes that are impacted by the update inside an isolated environment (Section 3.6).

5. Apply Patch. *gDSU* performs all the changes in correct order on the new environment. It also migrates the state of all affected live instances (Section 3.7).

6. Commit. *gDSU* replaces all the instances in the original environment affected by the update with their corresponding instances in the new environment. This step is only performed if the resulting environment is valid. If the validations are not correct, the new environment is just discarded and the update is not applied (Section 3.8).

7. Resume Threads. The application threads are finally resumed (Section 3.5).

We now explain in detail the different phases of the update process.

3.2 Patch Content

gDSU requires a patch to contain all the information necessary to perform the update in a single operation. The patch's content consists of:

Structural Changes. A set of all changes to methods and class structures corresponding to the new version. This includes new classes and methods, their modifications and removals.

Instance Migration Policies. A set of migration policies. A migration policy describes how to migrate live instances of one type from the current version to the new version. Migration policies and their implementation details are further explained in Section 3.7.

Update Validations. A set of validations. These validations are meant to guarantee that the application state and behaviour are consistent after the update is applied but before the commit.

A patch describes the update in a declarative way, containing details of what to do. The *gDSU* engine is responsible of determining when and how to perform the update.

3.3 Patch Generation

To help users using *gDSU*, *gDSU* provides several tools to help generate patches semi-automatically:

VCS version Difference. *gDSU* calculates a patch from two versions of the application in the version control system (VCS). This approach is useful when the update target system is an application in production.

IDE Change Sets. *gDSU* records IDE events and stores code changes. This approach is most useful in a development environment.

These tools automatically calculate the structural changes, system level validations and automatic migration policies. When the update is to be applied in a production environment the patch is generated in a development machine. This patch is later applied in the production machines.

Structural changes include modifications to methods, class structure and class inheritance changes. Structural changes are topologically sorted by their dependencies between each other. For example the creation of a given class is before the creation of a subclass or the definition of a method in the new class. As this order only depends in the static comparison of the versions and not in the live state of the application to be update, the sorting of the changes is done during the creation of the patch.

gDSU extracts the structural changes from two sources: (1) calculating the differences from two versions of the application and (2) recording the changes done by the developer. The former uses the capabilities of VCS solutions to provide a set of changes from one version to another. The later uses tools integrated with the IDE to extract the changes performed by the developer. Both methods return a set of changes to the elements of the application (*i.e.,* classes, methods), but the returned set could include different operations on the same element (*e.g.,* two versions of the same method or adding two instance variables to the same class) so these changes should be flattened (in the examples, taking the last version of the method and taking the definition of the class with both new instance variables) [DCD13].

gDSU automatically calculates migration policies based in the analysis of the changes. This static analysis allows *gDSU* to only identify changes in the structure that preserve the same set of values but ordered in different structure (*i.e.,* different order in the instance variables, instance variable moved to the super or sub classes). Automatic generation of complex migration strategies is analysed in Chapters 6 and 7.

As an example of an automatic migration policy, if a class' structure changes the order of its instance variables, the *gDSU* engine proposes a migration policy copying the instance variable values by name, as Listing 3.2 shows.

```
ByInstVarNameMigration >> migrateInstance: new fromOldInstance: old
                inNewEnv: newEnv fromOldEnv: oldEnv

    new class instanceVariables do: [ :newIV |
        old class instanceVariableNamed: newIV name
      ifFound: [ :oldIV | newIV write: (oldIV read: old) to: new ] ].
```

Figure 3.2: Reusable Migration Policy: it migrates all the instance variables' values by name. It is used when the instance variable order changes.

The developer can then extend the patch: modify the structural changes to apply, add business related migration policies (Section 3.7) and validations (Section 7.4).

The patch elements (*e.g.,* Migration policies, validatiosn) are reused in posterior updates.

3.4 Dynamic Patch Analysis

Before applying changes or perform any operation, *gDSU* performs an analysis of the patch and the state of the application to address the impact of the

patch in the running application. This analysis is used to validate the patch, calculate the classes of the live instances to migrate and the requirements to detect a safe update point.

In this stage, the patch is validated to guarantee that all the changes are applicable. The static dependencies of the changes are met. This means if a change requires an existing element in the environment, this required element exits in the environment to update previous to the update or is created previously during the update. As an example if a method is added to a class, the class should exist in the environment or it is created during the update.

Also the patch is analysed to calculate all the classes modified by the patch and which are the classes that require a migration policy. A class needs a migration policy if after the changes included in the patch its structure is modified. A class structure is modified if its own structure of variables is modified or the structure of the classes in its hierarchy is modified. A class with modifications only in methods does not require a migration policy.

The list of required migration policies is later used during the update. A migration policy is required to fulfil the update only if there are live instances of that class. If there are no live instances of that class, the migration policy is not required. In this stage of the execution of the update the live instances are not queried. This validation is later performed in following stages.

Finally this stage detects the modified methods, the list of modified methods are used to calculate a safe update point. By the definition of safe update point used by *gDSU*, there should not be active methods in the call stack of the threads that are modified by the update (Section 3.5).

3.5 Thread Management and Safe Point Detection

To guarantee isolation, *gDSU* suspends all the executing threads during the update window. This is required to guarantee that there is not any thread executing code that should be modified, also it guarantees that there are no new instances of classes that require instance migration.

Figure 3.3 presents a representation of how the update is performed from the threads management point of view. When the update is required, *gDSU* spawns a new thread that will perform all the operations related to the update. This thread is responsible of running the update code and will be the only one running during the update window.

Before executing the update, the *gDSU* thread executes a series of tasks. First, it executes the dynamic analysis of the patch, with this information it establishes the requirements for a safe update point. Then it monitors the all running threads (application and system threads) to check if a safe update point is reached. When a safe update point is reached, *gDSU* suspends all the

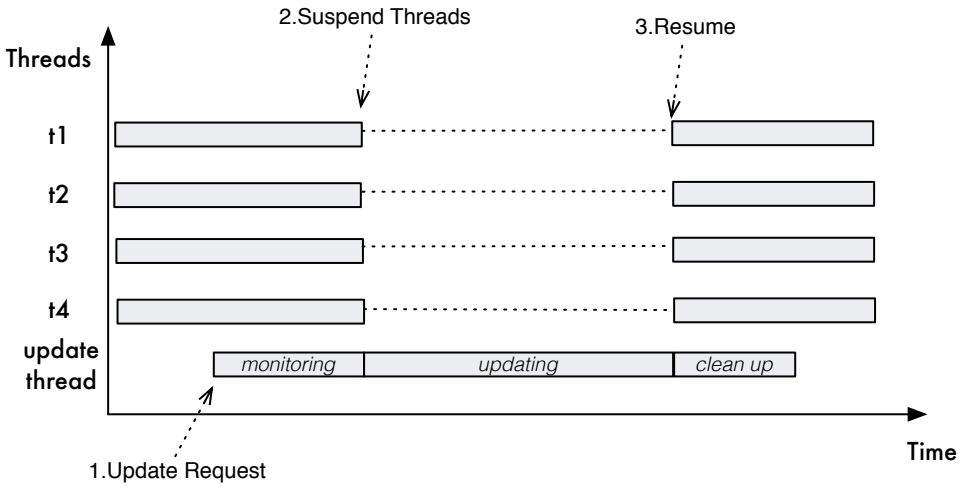


Figure 3.3: Threads Management during the update window

threads but the update thread. In this moment the update is applied. Once the update is finished, the suspended threads are resumed and the required clean up operations are executed. After the clean up is finished the update thread terminates.

Safe Update Point Definition and Detection

To avoid inconsistencies during the update and after the update, and to guarantee that the state of the application is correctly preserved and the application is not affected by the update, the update should be performed in a safe update point.

A safe update point is a point in time where the update is applicable in a safe way. At this point all the threads are suspended, and the update applied.

gDSU defines a safe update point as the moment in time when all the threads are at a safe update point. A thread is at a safe point if its call stack does not contain methods affected by the update *i.e.,* the corresponding patch contains changes for that method.

This conservative definition does not require call-stack handling or rewriting, as a result of the condition there is not method that is activated and modified by the patch.

To detect the safe update point condition, *gDSU* monitors all the running threads to check for such condition. The practical considerations in the implementation of this strategy are further analysed in Section 4.1.

3.6 Environment Copy

gDSU performs all the modifications in a copy of the environment, when the update is correctly validated it replaces the original environment with the new environment.

The environment includes the application state and system (live instances), the meta-state of the application and system (classes and methods) and the execution state of the application (threads, call-stacks).

Once all running threads are suspended, the update process copies the environment. Executing this copy in a practical way requires a series of considerations and techniques. Section 4.2 presents the practical details of the implementation of an efficient copy algorithm. In the implementation of *gDSU* only the modified elements are copied. This decision minimizes the length of the update window.

By using a copy of the environment *gDSU* is able to modify the elements in it, validate the changes and discard them in case of an error without affecting the original state of the application.

First, making a copy transforms the meta-circular update into a normal update because *gDSU* is not modifying itself but a copy of itself. Second, scoping the changes into a copy allows one to avoid affecting the updated application when problems appear during the update. Such problems can raise because of several causes *e.g.,* errors or bugs in the *gDSU* engine, the migration policies, or unsatisfied validations. While handy, the copy of the original environment is a time consuming operation, so limiting the number of copied objects reduces the overall execution time of the update.

gDSU implements a partial copy of the environment that only includes objects (classes and instances) affected by the update. *gDSU* calculates which objects are affected using the structural changes and the migration policies in the patch. A class is considered affected if there is a structural change on it or on one of its superclasses. An instance is considered affected if its class is affected or if there exists a registered migration policy for its class or any of its class' superclasses. Informally speaking, a structural change in a class will affect all the instances of that class and it will recursively affect its subclasses. As methods are not down-copied in subclasses, a change in a method only affects the class containing the method. The subclasses use the updated method through the usual method lookup.

This copy process leaves the original objects intact and thus requires to replace afterwards all references to the old objects with references to the new objects. In case of a valid update, the commit operation replaces the old environment with the new one. In case of an error or detection of an invalid update, the new environment is discarded and the threads resumed without

affecting the state of the application.

3.7 Application of Changes and Instance Migration

gDSU applies the update in the copied environment. All the modifications on classes and methods are applied on this copy of the environment. Changes do not affect the original elements in the original environment. Also, *gDSU* creates the new classes in this environment.

After applying the changes the new environment reproduce the classes and methods state of the application in the new version. Once, all the static state of the application, the migration of live instances should be performed.

To perform the migration of live instances, *gDSU* requires the definition of migration policies for each of the classes requiring instance migration. A migration policy is a first-class citizen that represents the migration logic from one version to the other of a given set of instances. This meta-level objects are used during the update and they are reusable in different versions and updates. Figure 3.4 shows the meta-object protocol of such a migration policy.

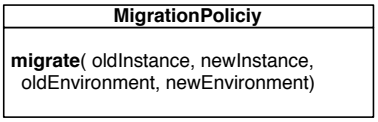


Figure 3.4: Migration Policy interface

The migration policy has access to the old and new instances of the element and the old and new environment. Doing so, the migration policy is able to configure the new instance with values coming from the old instance and from other instances of the new environment. For example, if a new instance needs to be created it should be done with the class in the new environment.

Figure 3.5 presents an example of a migration policy. This migration policy migrates the coordinates of the instances from a cartesian representation to a spherical representation.

The migration policies are implemented by the developers handle structural and application dependent changes. Examples of the former are when adding or removing an instance variable. Examples of the later is when an instance variable is used in different ways in different versions, *e.g.*, a instance variable contains a number and then it is replaced by an object.

```
Vector3DMigration >> migrateInstance: new fromOldInstance: old
                inNewEnv: newEnv fromOldEnv: oldEnv
    new radius: ((old x ** 2) + (old y ** 2) + (old z ** 2)) sqrt.
    new tetha: (z / new radio) arcCos.
    new phi: (old y / old x) arcTan.
```

Figure 3.5: Example of a manual migration

3.8 Validation and Commit of Changes

Once the changes are applied in the copy environment, *gDSU* validates this changes to guarantee the health of the application and the system. The results of the update are tested, if the update is correct the commit phase is performed; if the update is not correct the copy environment is discarded without affecting the original state of the application.

Validations are implemented as objects following the interface defined in Figure 3.6. These objects represent the invariants of the application and the system. There are two different sets of validations: system validations and application validations.

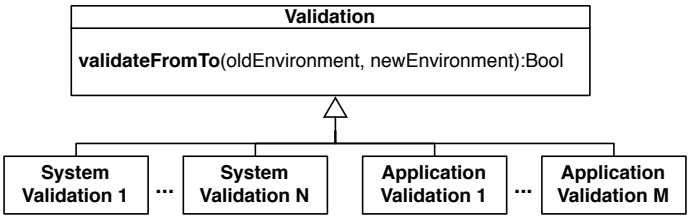


Figure 3.6: Validations as objects

System validations ensure that the invariants of the system are guaranteed. These validations are provided by the DSU solution. They validate that the system and its runtime are healthy after the update. Listing 3.7 presents an example of a system validation. This validation guarantees that the new created classes belongs to a package in the system. Packages are organizational units and each class in the system should belong to a package.

Application validations guarantee the application invariants. These validations are provided by the developer to guarantee the stability of the application. Examples of validations guarantee that: (1) all the products have a valid price, (2) all the students have a valid name, surname and student id. Listing 3.8 presents a possible implementation of the second validation example.

The validations have access to the old and new environment. These reifications of the environments give access to the static and dynamic elements in

```
PackageClassValidation >> validateFrom: oldEnvironment to: newEnvironment

    ^ newEnvironment allClasses allSatisfy:
        [ :aClass | aClass package isNotNil
          and: [ aClass package includes: aClass ]]
```

Figure 3.7: Validating that all classes have a proper package and the package includes the class

```
StudentValidation >> validateFrom: oldEnvironment to: newEnvironment

    ^ (newEnvironment classNamed: #Student)
        allInstances allSatisfy:
          [ :aStudent | (aStudent name isNotNil
            and: [ aStudent surname isNotNil])
            and: [ aStudent studentId isNotNil]]
```

Figure 3.8: Validating that all the students have the required information.

the environment. The validations are able to compare the state of the classes, methods and any live instance in the system. They access the live instances though the classes (*i.e.*, accessing to all the live instances of a class) or through the globally accessible variables in the environment.

Application validations require manual development. However, these validations are reused in different updates to the application and the system. During the life-cycle of the application the set of application validations are enriched with each update. Having more application validations increases the quality of the application, as it makes less probable that the application of an update that breaks the invariants of the application. They provide a similar advantage as the provided by a good set of unit tests.

Commit of Changes

Once the update is validated and the application is in healthy state, the changes should be applied in the original environment. During the commit operation all the elements that have been modified are replaced in the old environment with the their respective new environment version. This operation is performed atomically.

The whole heap and stack of the application is scanned to replace the references to instances in the old environment with the references to the corresponding instance in the new environment.

This operation already exists in Smalltalk systems, this operation is called *become*. This operation it is implemented at virtual machine level and it is already used to implement live programming. Similar operations are implemented in different VMs (*e.g.,* HotSpot JVM). If there is no operation present in the VM, it is implementable in the application level using proxies and bytecode rewriting.

Section 4.5 presents practical considerations to have an efficient implementation.

3.9 *gDSU* Platform Requirements

Implementing *gDSU* as a library requires that the underlying platform provides a number of requisites:

Class manipulation. *gDSU* needs to query, add and remove classes; and change the superclass. It also needs to be able to add, remove and change instance variables. Finally it needs to be able to add, remove and modify methods.

Instance manipulation. *gDSU* needs to create new instances from a given class, read and write all the instance variables in a given instance. And also, it needs to list all the instances of a given class.

Thread manipulation. *gDSU* needs to inspect the call stack of threads, stop and resume them. It needs to be able to modify the call stack inserting new method activations.

Environment manipulation. *gDSU* needs to read and modify the elements in the global environment. It needs to make a copy of the environment and replace the environment with this copy.

Bulk object swap. *gDSU* needs to perform a bulk replacement of objects as described in the Section 3.8.

IDE Integration. To to minimize the manual building of patches, *gDSU* needs to integrate with the language IDE. *gDSU* needs a way of getting the changes performed by the user and the details of the program modified.

3.10 Conclusion

This chapter presents *gDSU*. *gDSU* is a DSU solution that is applicable in production and development environments. This general solution covers all the requirements established for a DSU solution in Chapter 2 . This chapter presents the solution and describes all the steps in the execution of the update.

Each of the steps of the execution explains its design and how they are integrated in the big picture of the solution. The different steps in the execution of the process map to the requirements for a general DSU solution.

Designing Techniques for an efficient gDSU

Chapter 4

Contents

Implementing a practical DSU solution requires solving practical problems. To have an implementation that is useful for users in both environments, the implementation should be efficient and perform the updates in a small update window with the smallest possible memory footprint.

In the implementation of *gDSU* a number of techniques have been developed to implement it in a practical way. This chapter presents the required techniques and their details. These techniques include the detection of a safe point update (Section 4.1), and an efficient way of copying an environment (Section 4.2).

Also, this chapter describes the implementation and reuse of migration policies and validations (Section 4.3 and 4.4). Finally, it presents the required language and virtual machine machinery needed to implement *gDSU*: Bulk Instance Replacement (Section 4.5) and a Modular Class Installer (Section 4.6).

4.1 Automatic Safe Update Point Detection

gDSU guarantees that the updated application is not running during the update by suspending all its related threads. Process suspension contributes to the atomicity of the update. To provide better guarantees and avoid creating execution inconsistencies, the application should not be suspended at any moment but at a point considered safe. During the update process the only running thread is the update engine thread.

As defined in Chapter 3, an application is at a safe point if all its threads are at a safe point. A thread is at a safe point if its call stack does not contain methods affected by the update *i.e.*, the corresponding patch contains changes for that method.

When an update is required, the *gDSU* engine spawns a new thread. This thread is responsible of performing the update. Then the *gDSU* engine waits until all other threads are at a safe point before performing the update. The update thread monitors all other threads using events rather than busy-waiting. When a thread returns from the execution of an affected method, an event is produced. Upon an event, the update thread checks if a safe point is reached, in which case starts applying the update. If the application is not at a safe point, the update thread yields the processor and waits for the next event to recheck.

gDSU implements such a detection strategy with call-stack manipulation. It inserts in each thread call-stacks one notification call context just before the context of oldest affected method in the stack. When the notification call context is executed because the thread returns from the affected method, it suspends all the running threads and re-checks for the occurrence of a safe update point.

While checking for safe update point condition *gDSU* will insert new call contexts in all threads that do not have one. This is required because existing threads may have returned from the notification call stacks or new threads include affected methods in their call-stack.

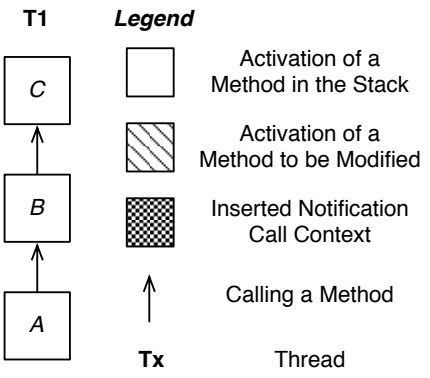
Figure 4.1 presents an example of the detection process. First, the threads with modified methods should be identified. To perform so, the call-stack of all the threads are analysed (Figure 4.1b). In this state the update cannot be performed. The process should wait until there is no modified method activation left in any thread.

To detect a safe update point, the call-stack is modified to insert a context between the caller of a modified method and the modified method (Figure 4.1c).

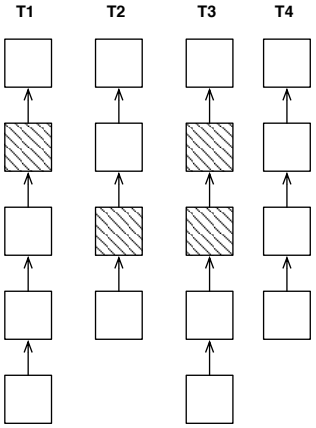
Each time one of these inserted contexts is executed it activates the update thread to detect if a safe update point is reached. If the system is not in a safe update point, the update thread will yield the processor and the application will continue executing (Figure 4.1d). In the case of new offending threads are detected, new detection contexts are added.

If the update thread is activated by a detection context. And the update process detects that the application is in a safe update point. The update is performed (Figure 4.1e).

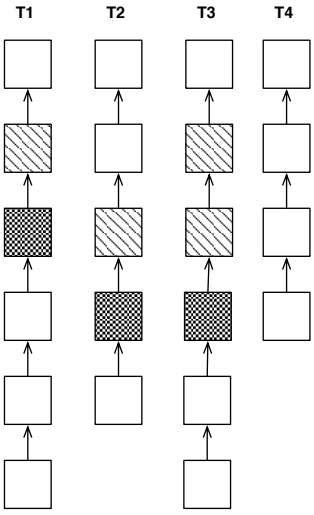
This conservative strategy may not converge if the application never reaches a quiescence point [NH09]. *gDSU* will timeout an update after sev-



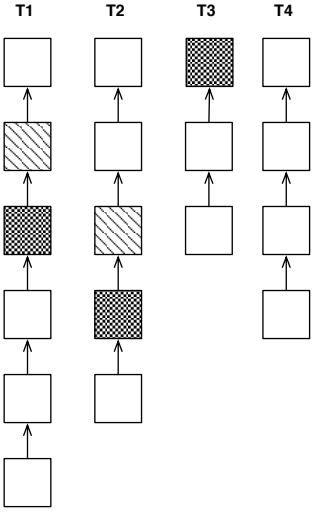
(a) Method A called B, and B called C.



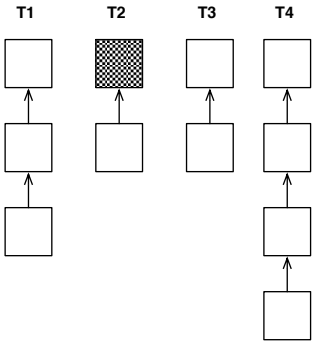
(b) Threads using methods to be modified, in this state the update cannot be applied.



(c) Context inserted in the call stacks just before the methods that should be updated.



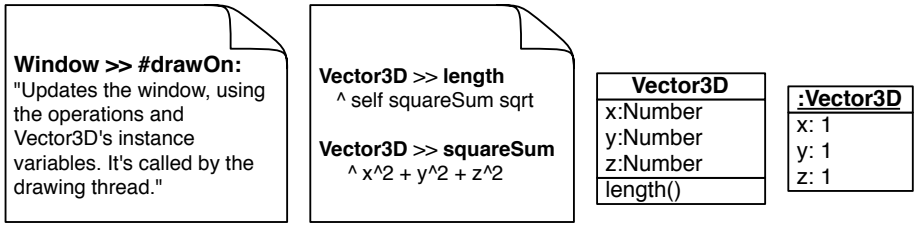
(d) The *gDSU* checks, but the system is not at a Safe Update point because T1 and T2 are still executing methods to be modified.



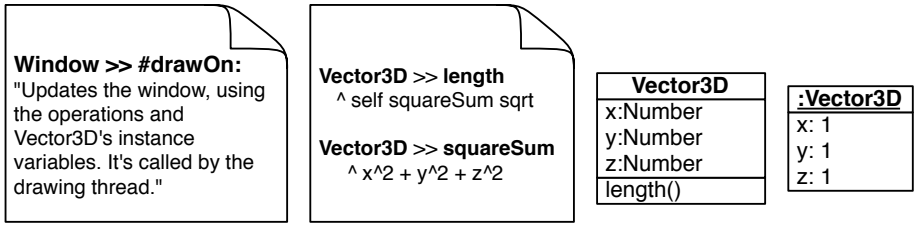
(e) The *gDSU* checks, the system is at a Safe Update Point. The update is performed.

Figure 4.1: Modification of call stack for the detection of Safe Update Points.

eral of retries, aborting the update. The reasoning of this choice is discussed later.



(a) Original version using cartesian coordinates



(b) Modified version using spherical coordinates

Figure 4.2: Example of a modification requiring safe point detection

In the example presented in Figure 4.2, the problematic thread is the drawing thread which is calling the drawOn: method. If during the safe point detection the drawing thread is executing the drawOn: method, the *gDSU* engine adds a notification call context just before this method. When the drawOn: method ends, the thread executes the notification call context and notifies the *gDSU* engine that is possible to suspend all threads and execute the update.

Conservative Safe Update Points

We decided to look for safe update points instead of reconstructing the threads' call stack after the update. This decision simplifies the solution and permits the execution of any change in the instance structure or in the methods. Our solution does not stop the execution of the application while it is waiting for a safe update point, the application is running freely until this safe point is reached.

Rewriting the call stack allows the execution of the update in any moment, without needing to wait for reaching a safe update point. However, stack reconstruction techniques are limited in the number of method changes it can handle without developers' manual intervention, specially with loops and recursive methods. They also require full instrumentation of the code and enough history of previous runs.

Our solution provides a way of automatically detecting safe update points in a pretty conservative manner. We are aware that the proposed safe point

detection does not always arrive to a safe point, since some programming patterns produce programs that never reach a safe point in our definition. For example, patterns such as the one in Figure 4.3 produces a method that never returns so it never reaches a safe update point.

```
execution
        [ true ] whileTrue: [
                self doAction1.
                ...
                self doAction2.
        ]
```

Figure 4.3: Example of a Programming Pattern that does not allow the program to reach a safe update points if this method is updated in the patch.

We have decided that aborting the update is better than performing the update without the guarantees needed to continue normal execution. Less restrictive conditions are possible, but these conditions are more complex to detect and they do not provide enough guarantees to execute all the possible changes. Other solutions [PH13, MME12] require explicit update points to handle these situations.

4.2 Efficient Partial Copy of the Original Environment

gDSU makes a partial copy of the original environment into a new environment to guarantee that it can safely perform self-updates, core library updates and cancel failing updates. The copy of the environment is performed during the update window, so the duration of the update window is directly related with the duration of the copy. So, the amount of copied objects should be minimized to minimize the time.

gDSU copies the following objects:

- Objects that represent the environment. These elements are the representation of the environment in the reflective language. These elements are modified when a class is installed in the system. They should reference the new classes in the copied environment. In Pharo, these elements are the global accessible environment (all the global bindings), the package manager state (Representing the organization in packages of the classes) and the class organizer.
- Classes modified during the update. The classes are detected on usage. Only the modified classes are created in the system.

- Instances affected by the changes in the modified classes. These instances are identified using the set of classes modified in the update.

4.2.1 Detection of Modified Classes

The detection of modified classes is performed during the execution of the update. The classes are only copied before the modification. Moreover, new and modified classes are created in the new environment. As a class is modified, it is created directly in the new environment with the changes already applied.

gDSU handles the need of copy the related classes of the modified class. These related classes are the super class and subclasses of a modified class. The superclass is not copied as modifications in the subclasses does not affect the superclass. The subclasses of the modified class are recreated in the new environment if the modification of the class affects its structure. For example, only modifying the code of a method in a class does not require to copy its subclasses. On the other hand, a modification to the structure of a class (adding or removing an instance variable) requires the copy of all the subclasses as the subclasses structure depends in the structure of the modified class.

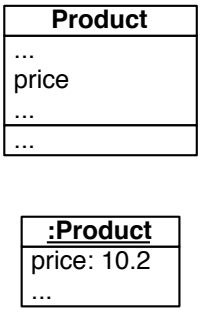
4.2.2 Detection of Instances to Migrate

During the update process *gDSU* creates a copy of live instances requiring migration. All the instances of classes with modified structure should be migrated. Also, all the instances of subclasses of these modified classes should be migrated.

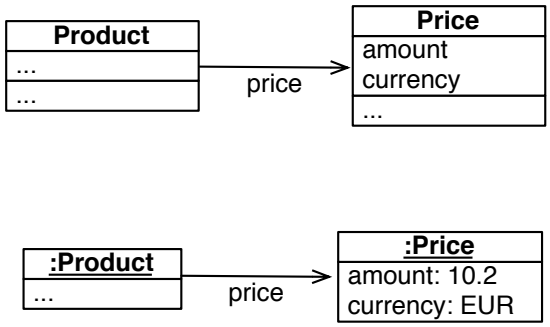
There is a set of instances that should be migrated even though there is no modification in their structure. If there is a modification in the value of an instance variable it should be migrated. If there is a business logic modification that changes the use of an instance variable, the instances should be also correctly migrated.

Figure 4.4 presents a modification introducing a price object that requires the migration of Product instances, even though the class structure is not modified. In this example the value stored in the price instance variable changes from using a Float value to a reference to a Price object that encapsulates the value and the currency.

The requirement to migrate instances of classes affected by a business-logic change is performed by checking if there is a migration policy for this class. So, if there is a migration policy for a given class all the instances of that class and its subclasses are migrated applying the migration policy.



(a) Original Version: using a float to represent the price of a product



(b) Updated Version: using an object to represent the price of a product

Figure 4.4: Update introducing an instance migration for a business-logic change.

4.3 Reusable Instance State Migrations

Migrating the state of instances from one version to another requires the developer to implement migration logic in the form of migration policies. Figure 4.5 shows the meta-object protocol of such a migration policy.

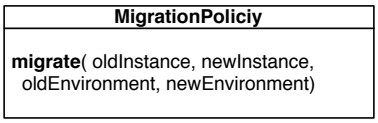
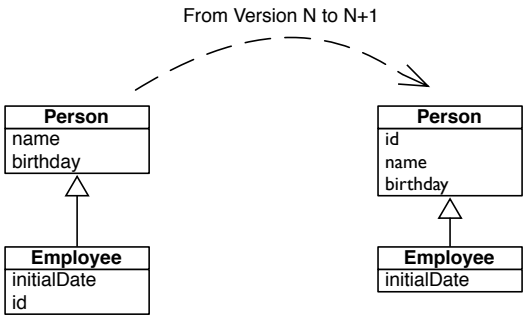


Figure 4.5: Migration Policy interface

To ease the effort to implement migration logic *gDSU* provides generic migration policies for common cases such as *e.g.,* refactorings. These generic migration policies are reused between different updates and even between different applications. Figure 4.6 shows a pull-up instance variable refactoring and the corresponding migration policy that is applicable every time the same refactoring is applied.

Moreover, the developer may extend this protocol to define business dependent migrations. For example, Figure 4.7 illustrates a policy to migrate cartesian to spherical coordinates.



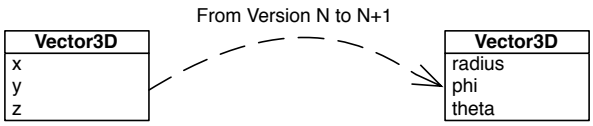

```
PullUpMigrationPolicy >>
migrateInstance: new fromOldInstance: old
inNewEnv: newEnv fromOldEnv: oldEnv
        new class instanceVariables do: [ :newIV |
          old class instanceVariableNamed: newIV name
                                                    if-
Found: [ :oldIV | newIV write: (oldIV read: old) to: new ] ].
```

Figure 4.6: Migrating instance variables per name: an example of application independent change

In our solution the developer has to implement part of the patch *i.e.,* migration policies and validations. We have decided to do that because most of these components are heavily coupled with the application under update. However, the *gDSU* tool is shipped with already implemented versions of such elements. The shipped implementations cover the common scenarios, and provide a way of extending them. The shipped system validations assert the correct state of the Pharo environment after an update. We consider the validations and the migration policies as elements that are equivalent to unitary tests. The required elements are part of the application code base and they are created during the whole life cycle of the application.

4.4 Reusable Validations

To guarantee that an application state and behaviour are consistent after an update, *gDSU* performs several *Validations* before committing the update. A *Validation* is a predicate function that validates the copied environment. If all validations are successful *gDSU* proceeds with the commit, otherwise the update is discarded. Although the validations are not needed by the update engine, their presence improves the stability of the updates avoiding invalid updates. We identify two different categories of validations that are easily reused in different updates.




```
CartesianToSphericalMigrationPolicy >>
migrateInstance: new fromOldInstance: old
inNewEnv: newEnv fromOldEnv: oldEnv
    new radius: old length.
    new thetha: (old z / new radius) arcCos.
    new phi: (old y / old x) arcTan.
```

Figure 4.7: Migrating Vector3D: an example of application dependent change.

System Level Validation. It checks the consistency of the running platform. They are independent of the update and the application, and they are executed in all the updates for a given platform. For example, one validation checks if the application meta-objects (classes and methods) and system structures have been correctly migrated (*e.g.,* the inheritance relationship is maintained for classes and metaclasses).

Application Validation. It checks application invariants that should be consistent during all the life-time of the application. They are applied in all the updates of this application and are useful to guarantee business rules before committing an update. It is the responsibility of the developers to produce them. For example, an application validation may check that all Employee instances have a name.

Using the same validation mechanism, *Single Update Validations* are implemented. This validations check a condition that should hold when the update is applied. For example, if the structure of a core object is changed, it is useful to check if the state migration was correct for every object. Once this update is committed, this validation is not useful anymore and can be discarded. In the running example presented in Figure 4.2, the developer includes a validation that asserts the correctness of all the points to prematurely detect and avoid problems in the drawing thread. Figure 4.8 shows an example of such a validation.

Single update validations guarantee the correctness of a given update. However, reusable validations provides not only guarantees for a single update but for all the following updates. Moreover, they constitute a documentation and quality element for a given application in the same way as unit tests do.

```
CartesianToSphericalValidation >>
validateFrom: oldEnvironment to: newEnvironment
    ^ (newEnvironment allInstancesPairFor: #Vector3D)
        allSatisfy:[:aPair | | old new |
            old := aPair first.
            new := aPair second.
            (old length = new length) & (new phi = (old y / old x) arcTan).
        ].
```

Figure 4.8: This validation is used to guarantee the correct migration of the Vector3D from one coordinate system to the other.

4.5 Bulk Instance Replacement

gDSU implements the commit operation as a bulk swap of object references. In other words, during the commit operation *gDSU* replaces all references to old affected objects by references to their corresponding copies. Bulk replacement is done atomically, making the update a true atomic process. The bulk replacement operation is crucial in the implementation of most DSUs. They are used in related works such as Rubah [PH13] and are present in most Smalltalk implementations. However, a naive implementation would not be satisfying from a performance point of view as it would require to full scan the memory to perform pointer replacement [MB15].

Bulk replacement is implemented in our prototype as the primitive Virtual Machine operation become:. This operation takes two equal-sized arrays as arguments and performs a pointer-swap between each object in the first array and the object that occupies the same position in the second array. That is, a pointer swap between a and b makes all objects in memory referencing a change and point to b and vice-versa. For the atomic commit, *gDSU* uses a variant of become: called becomeForward:. This operation, also called one-way-become, only performs the pointer replacement from the first to the second set of references. To efficiently implement become we leverage a novel memory management technique called forwarders that is available in the Pharo Virtual Machine. Forwarders allow one to perform lazy pointer swapping with the use of a partial read-barrier thus avoiding the need of a memory full scan [MB15].

We have decided to use eager instance migration instead of lazy migration because lazy instance migration requires the use of proxies. By using eager migration, the penalty during the normal execution of the application is zero. During the normal execution of the application there is no need to execute

code of the DSU solution.

4.6 Extensible Class Building Process

The creation of classes in running application is divided in three different stages:

Class Creation. During this stage the class is created in the system. In a reflective language the classes are other objects in the environment. The creation of the class requires a series of steps correctly executed to produce a valid class that is later used to create instances. The correct creation logic handles the creation of the class, its metaclass and all the related objects (*e.g.*, method dictionary, instance layout).

Class Installation. After the creation of the class, the new class should be correctly installed in the system. This registration includes other operations such as setting the global reference to the class, registering the new class into its superclass, adding the class to the corresponding package.

Instance Migration. Once the class is installed in the system. Live instances of the old class should be migrated to the new class. This migration of instances is performed without caring about the state of the object. Only the matching instance variables are preserved. New instance variables are not initialized leaving them referencing null.

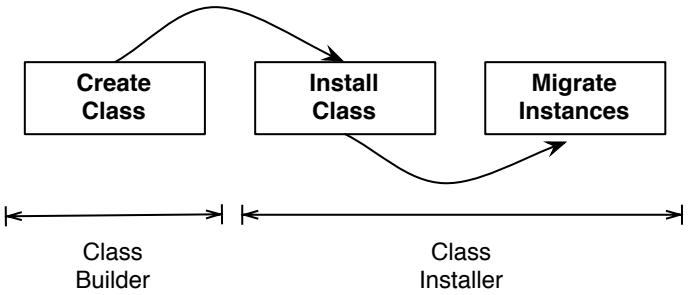


Figure 4.9: Class Building and Installing Process

In Pharo, these stages are handled by two components of the system: *Class Builder* and *Class Installer*. Such component is already present in the environment and performs these stages correctly. Figure 4.9 shows the stages of the creation of a class and how these two components perform the different stages.

The actual implementation in Pharo is correctly handling the creation and installation of classes. However, implementing a DSU solution also requires

to perform these stages during the application of an update. Also, the current implementation does not present a clear separation of the class builder and class installer responsibilities limiting the ability to reuse the components.

One possible solution is to reimplement all the behaviour implemented in the class builder into the DSU solution. This solution is not optimal as it duplicates the logic in two different components. This duplication increases the maintenance work required when there is a modification in the class creation and installation logic.

To correctly integrate a DSU tool into Pharo, we developed a modular implementation of the class builder and installer. This new modular class builder and installer is called *Shift*.

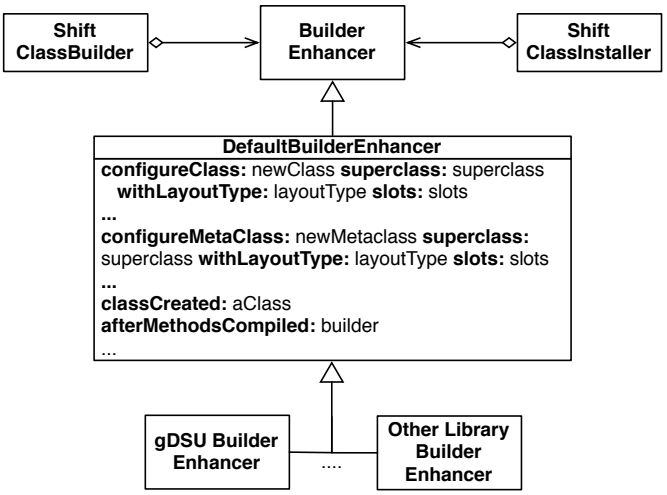


Figure 4.10: Shift Builder Enhancers

Shift class installer and builder expose different extension points during the three stages. They share a common extension point the BuilderEnhancer. Libraries requiring to extend the behaviour in the creation of classes are able to collaborate with different builder enhancers.

The builder enhancer provides an interface of integration for all the steps in the class creation process, also it provides a way of replacing the target environment, the classes to use and all the logic during the creation and installation process. Shift provides a default implementation that includes the default behaviour of the class builder and installer.

Each library implements a builder enhancer to contribute during the process of creation. Moreover, the builder enhancers are composable opening a way of integrating different libraries and tools in a decoupled way. Figure 4.10 shows the builder enhancer hierarchy.

Finally, Shift also provides a clear interface to divide both steps of the process. It provides a clear division of the class builder and the class installer. Both components are usable independently.

gDSU not only uses the class builder and installer, but it also extend it through the implementation of builder enhancer. This is done to control the following aspects of the class building:

- Changing the target environment of the classes.
- Detecting the need of creating in the new environment a superclass or subclass of a modified class.
- Interrupting the default instance migration as the migration is later performed by the DSU tool.
- Registering the modified classes and subclasses to create corresponding automatic validations and migrations of instances.

4.7 Conclusion

This chapter presents the required techniques and machinery to implement *gDSU* in a practical way. A DSU solution is practical if it minimizes the manual work from the developer, performs in an efficient way and it is usable in both production and development environments. The techniques and machinery presented in this chapter make possible to implement the concepts of *gDSU* in a practical way. By using the presented techniques, *gDSU* achieves the benchmark results shown in Chapter 5.

Also, this chapter presents the design decisions that we have taken in the implementation of the required techniques. The set of practical solutions presented in this chapter, even thought they are tailored to be used by *gDSU*, are applicable in other practical solutions.

For example, the extensible class building process has been used to implement the support for implementing reuse mechanisms called *Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM* [TPF+18a] and modular stateful traits [BDNW07] and talents [RGN+12] implementations for Pharo.

Validation of gDSU for production related requirements

Chapter 5

Contents

To validate *gDSU* it was implemented in Pharo [BDN+09]. Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback[1]. It has been implemented in Pharo because (1) it provides powerful meta-level operations, (2) most of its runtime is implemented in itself, (3) it represents basic concepts of the language and the environment as first-class objects. These characteristics allow the implementation of *gDSU* as a library. Moreover, Pharo covers all the platform requirements to implement our solution (Section 3.9).

This prototype is available in a Git repository[2]. It is loadable in the latest stable version (Pharo 6) of the platform and it is intended to be included in future Pharo versions.

gDSU is applicable not only for updating a running application, but also in daily development. It is validated in the following three scenarios. Each scenario has been validated as a long running application and using interactive live programming:

- Update of Application Code with live instances (Section 5.2).
- Update of the DSU engine itself (Section 5.3).

[1] http://pharo.org/
[2] https://github.com/tesonep/pharo-AtomicClassInstaller

- Update of the core libraries of the language (Section 5.4).

gDSU also has been validated using a set of benchmarks. These benchmarks show that the solution is viable in terms of execution time and used memory (Section 5.5).

Finally, in Section 5.6 we analyse the requirement assessment of *gDSU* for a production DSU

5.1 Validation Set-up

The following sections present three different scenarios we used to validate and benchmark our solution. These scenarios include the modification and refactoring of a stateful chat application. The modifications are performed in the application, the DSU and in the core system libraries. For each of these scenarios we applied the following set-up for the development and production environment:

Development Environment. We set up a development environment with our application code. We run a simulation of requests to generate application objects. This simulation produces around 30 000 live instances. After the simulation, we have an environment with live instances that is useful to perform live programming. This environment replicates a common development environment where the developer has not only the code but also a set of data to try her changes. Then, we perform modifications to the application. These modifications are performed programmatically, but the result is the same if they are produced using the existing IDE.

Production Environment. Using the same application a HTTP server is launched. This HTTP server replicates a production server. This is designed to be deployed as a productive application, as it uses the production ready frameworks and technologies used in Pharo. We generate 10 threads with loops sending requests to our server. They run concurrently during 2 minutes, generating an average of 700 requests per second. This simulation generates the load expected in a production server. Then, we apply the update at minute 1.

Finally, we apply measurements after the update is finished.

The scenario application, the set of test scripts and detailed instructions are available in GitHub[3]. Also, Appendix B describes detailed instructions to replicate the validation experiments.

[3] https://github.com/tesonep/chatServer.git

5.2 Validation 1: Application Update

Research Question. Is *gDSU* able to safely update a running stateful application in a development or production scenario?

Scenario. Our scenario application is a chat application. This application stores all the messages sent by all the users generating live instances in each request.

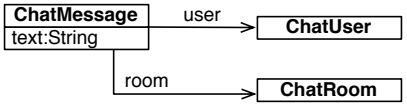


Figure 5.1: Original design. All the messages are instances of a single class ChatMessage. This implementation has conditional code to handle the differences in messages from the system and from users.

Figure 5.1 shows the model of the application in Version 1. For presentation purposes, we show only the part that is relevant for the update. The application handles two types of messages. The first type is the one sent by a user in a room, the second type is the one produced by the room (*e.g.*, when a user enters or leave). In this version all the messages are instances of ChatMessage, when the message is an info message (that is not produced by a user) the user field is left as null.

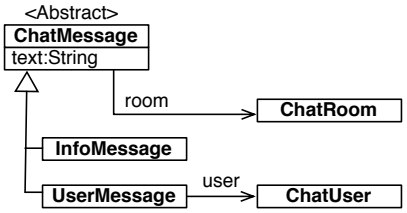


Figure 5.2: The application is refactored to extract the different behavior in the messages in two subclasses (InfoMessage and UserMessage) to represent the messages sent by the system and by a user.

Figure 5.2 shows the model of the application in Version 2. In this version the ChatMessage is refactored to include two subclasses. One for the user messages and the other for the info messages. Going from a version to the other requires the migration of live instances.

Results. *gDSU* updates correctly both the development and production applications. Instances are correctly migrated in an atomic fashion. No inconsistencies are introduced.

5.3 Validation 2: Update of the DSU

Research Question. Is *gDSU* able to update itself?

Scenarios. We have performed three updates on *gDSU* code:

1. Update the safe point detection algorithm.
2. Update the application of structural changes.
3. Update the internal representation of the update (*i.e.,* modifying a *gDSU* stateful class with live instances during the update).

Results. All updates were successful. All of them show that *gDSU* can update code and migrate instances that are related to and used by itself. Moreover, the first experiment shows that the safe update detection works even if the affected method is in the DSU thread.

5.4 Validation 3: Update of Language Core Libraries

Research Question. Is *gDSU* able to update core language libraries?

Scenarios. To validate the ability to update the language core libraries we experimented with the following two scenarios:

1. Update the OrderedCollection class, adding a new instance variable holding the size of the collection and modifying all related methods. The OrderedCollection class is a key part of Pharo's collections framework. It is the main collection used in the whole environment. In any given Pharo environment there around 46 000 live instances of this class. Also this class is extensively used by *gDSU*.
2. Update the class builder modifying the methods *gDSU* uses to create classes. All the operations modifying a class in Pharo are performed through the class builder. This component is a crucial part of the live programming capabilities of Pharo. Also, it is used as a key part of *gDSU*.

As said before, both elements are used during the execution of the update, introducing circularity issues similar to the changing *gDSU* itself.

Results. All updates were successful. These experiments demonstrate that the running core libraries are indeed isolated from the update. Otherwise, modifying the core libraries while performing the update risks to compromise the whole application stability.

5.5 Validation 4: Benchmarks

To evaluate the performance of our solution we have performed two series of benchmarks. The first one analyses how the number of live instances to migrate affects the update time. The second benchmark analyses the impact of the update process on the response time of a running application. The benchmark has been executed using Pharo 6.1 32-bits, in a machine running OS X 10.12.6 having a 2,6 GHz Intel Core i7 and 8 Gb of 1600 MHz RAM memory.

Number of Migrated Instances. This benchmark shows the behaviour in terms of memory and time with a varying number of instances to migrate. For this benchmark we use the server application of *Validation 1*. This scenario allows us to have different quantities of instances to migrate after the change. We performed 172 updates varying zero to ten million instances and we analysed the time and memory required to perform the migration. The results, in Figure 5.3, show that (1) the memory consumption is linear and it grows with the number of instances to migrate and (2) the time to execute the update is almost constant (around 1 second) below ten thousand instances and then it grows linearly.

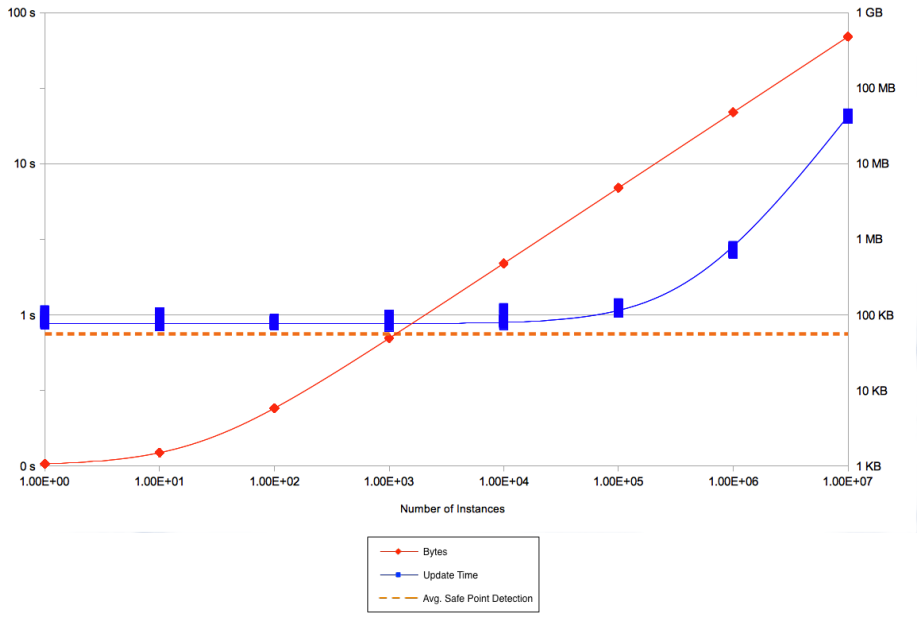


Figure 5.3: Impact in memory space and execution time depending the number of instances to migrate.

The memory consumption linearity is due to the copy of the affected instances. The DSU process copies the modified instances and classes to perform the changes. The number of affected instances has a baseline of 13 in-

stances when there are no live instances of the modified class to migrate. This set of instances are the core objects modified in the system *i.e.,* package manager, package, classes, global environment. So, any change will include at least these 13 instance to migrate. These instances are the minimum affected by an update. Starting from this baseline the update process only copies the objects affected by the update.

The variation in the execution time is due to two causes. The first one is that the detection of a safe update point takes an average of 750 milliseconds. During this time the application is running normally and the update process is just waiting. The second cause is that the bulk replacement takes a constant time of 250 milliseconds in average to create the forwarders. If the number of forwarders does not fit in the available free memory, the bulk update runs a garbage collection and performs a traditional pointer swapping with a full scan [MB15]. Our benchmark shows that bulk replacement is able to handle about 25 thousand instances per second.

Server Response Time. This benchmark shows time measurements on the server application described in Section 5.1. The results, illustrated in Figure 5.4, shows the application response time. The response time during the update windows is of 1400 ms, with a number of instances to migrate around 28500. Contrastingly, the response time outside the update window ranges from 5 to 200 ms, with an average of 22 ms. This benchmark shows that the impact in response time is inside the parameters of the first benchmark.

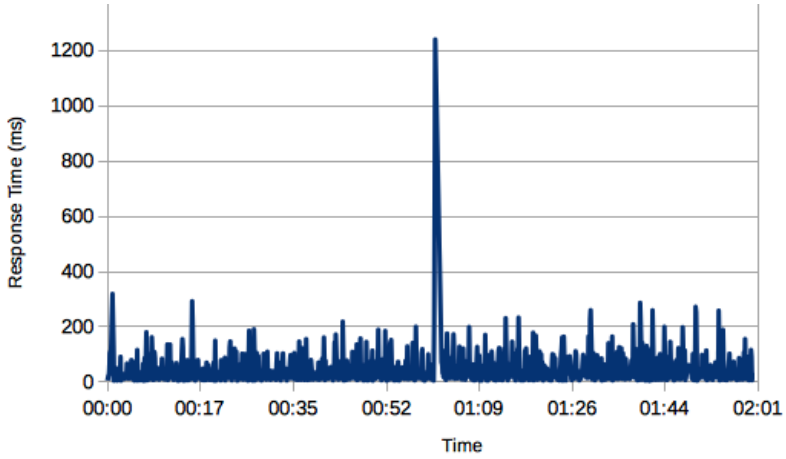


Figure 5.4: The response time is only affected briefly during a small update window.

5.6 Requirement Assessment

This section analyses how *gDSU* satisfies each of the stated requirements. We are analysing the capability of *gDSU* to be used as a production DSU.

Atomicity. We satisfy the atomicity requirement by applying the changes in an isolated environment, and committing all the changes in a bulk replacement operation (Section 3.8).

State Migration. We satisfy state migration requirement through the use of migration policies (Section 3.7). The developer provides the required migration logic implementing one or more migration policies. The migration policies are reused in different updates and also the implementation provides generic migration policies.

Automatic Safe Point Detection. We satisfy safe point detection requirement by the implementation of an automatic detection algorithm based on stack manipulation. The developer does not need to provide extra information to detect safe points (Section 3.5).

Isolation. While the changes are not committed, they are isolated into the alternative environment. The instances and classes of different versions are never mix up (Section 3.8).

Patch Generation. We satisfy the patch generation requirement with a semi automatic patch generation (Section 3.3). Our proposed patch generation uses two strategies: (1) getting the information from the version control system and (2) storing the information while the developer changes the application. The first strategy is used when the patch is generated to migrate one version of the application to another. The second strategy is used when the application is modified using live programming. In both strategies, the patch is not totally generated. The developer should provide the business related migrations that are impossible to calculate.

Patch Reuse. Our proposed solution requires the implementation of custom migration and validation logic. However, the solution provides different mechanisms to reuse and combine this custom logic (Sections 3.7 and 7.4).

Self Update and Core Lib Update. Our solution satisfies the self update and core libraries update requirement using an atomic commit operation. Our solution performs all the changes in a copied environment. This level of isolation allows us to modify elements that are currently used by the update

process. The real elements are replaced in a single operation using a bulk instance replacement mechanism (Section 3.8).

Small Run-time Penalty. We minimize the run-time penalty through the usage of eager migration of instances. Using eager migration does not require the use of lazy proxies or bytecode instrumentation. Our solution does not add any impact in the execution outside the update window. Moreover, no part of the *gDSU* engine runs outside the update process. We validate this in Section 5.5.

Minimal Application Downtime. Our solution minimizes the downtime of the application by minimizing the copy of objects. Only the modified parts of the application are copied and replaced in an update. The efficient copy of updated elements apply to live instances, classes and methods (Section 3.6). We validate this in Section 5.5.

5.7 Conclusion

In this chapter, we present a validation of *gDSU*. This validation has been implemented in Pharo and it has been tested in different scenarios. These scenarios where designed to show that the proposed DSU solution complies with the requirements for a DSU that is designed to be used in production.

Table 5.1 presents a summary of the comparison of our proposed solution and the production DSU solutions.

Category	**Requirements**									
	Atomicity	State Migration	Automatic Safe Point Detection	Small Run-time Penalty	Minimal Downtime	Isolation	Patch Generation	Patch Reuse	Self and Core Update	**Examples**
Production DSUs	●	●	◐	○	●	●	M	●	◐	Rubah, DuSTM, Pymoult
gDSU	●	●	●	●	●	●	S	●	●	

●: Yes ○: No ◐: Limited
A: Automatic M: Manual S: Semi-Automatic

Table 5.1: gDSU vs. Production DSU

Also, this chapter presents a benchmark of the solution implemented in

Pharo. This performance validation is included to show that the implementation represents a practical DSU solution.

Part III

DSU for Live Environments

Atomic State Preserving Refactorings

Chapter 6

Contents

Refactorings are behaviour preserving operations that help developers to improve the design of an application [Fow99, RBJ97, DHL96]. These code transformations modify the implementation of the application keeping its features. They improve the overal quality of the application [RBJ97]. Nowadays, refactoring tools are present in the majority of Integrated Development Environments (IDE) used in the industry [MT04], but with different degrees of refactoring supports. A refactoring is composed of pre and post-conditions as well as a number of ordered elementary steps. Each step modifies the classes and methods. Automatic refactorings constitute a daily tool used by programmers to improve the quality of their code [KZN12, XS06, DJ05, KZN14, BDLDP+15].

However, depending on the performed change, the internal state of live instances may be corrupted, making them unusable for the running program. For example, adding an instance variable initializes the new instance variable to *null* for all live instances. Such bad initialization may break program execution. As explained later, there are other refactorings leading to such *instance corruption*.

The use of a refactoring tool in a live programming environment amplifies the *instance corruption* problem. We show that 36% of the refactorings described by Fowler *et al.* [Fow99] present this problem when they are applied in a live programming environment (cf. Section 6.1).

We propose an implementation of automatic refactorings that takes advantages of using a DSU tool. Each refactoring runs in a transaction and it

affects all instances and classes in an atomic fashion. For each refactoring requiring migration of instances, a migration strategy is provided. By doing so, the instance corruption is removed when applying automatic refactorings.

In this chapter we present: (1) an analysis of the impact of refactoring tools in a live programming environment. (2) a new technique (using *gDSU*) for applying refactorings in live programming environments. This technique allows developers to perform refactorings while preserving the state of live objects and thus the correct behaviour of the running program. And (3) a validation of our proposed solution through an implementation in Pharo [BDN+09] using *gDSU*.

6.1 Class Refactorings that break Instances

Although it is possible to perform refactorings by hand, tool support is crucial to increase productivity [TB01, MT04]. Refactoring tools guarantee software behaviour consistency while preserving its correctness [RBJO96]. However, this guarantee is not extended to live instances that constitute the runtime environment. Live instance correctness is crucial when doing live-programming, as the program is executing while the modification is performed.

A refactoring operation involves a number of small modifications of the code and the structure of the objects. These operations are usually performed sequentially, modifying classes one change after the other, without handling the refactoring as a complex atomic change. Since the scope of default refactoring tools is static (i.e., they manipulate models of the code not of the instances), they focus on preserving a correct behaviour. However, a problem arises when refactorings are applied in a live programming environment. Indeed, live objects whose classes were modified should be migrated from the previous structure to the new structure. This need of migrating instances is not addressed by existing refactoring tools as they are not intended to be used in an environment with live instances.

As an extreme example, in bootstrapped and reflective systems [PDF+14, PDFB15], applying a refactoring on system classes may result in an instability of the whole system if instances are not correctly handled by the refactoring tool. This is why developers must carefully plan transformation steps to preserve the internal state of "kernel" objects [PDF+15]. This issue is present in reflective languages such as Pharo, Self [US87], Newspeak [Bra07, Bra10] or Strongtalk [Str]. These environments allow the developer to change all the elements without differencing application, core libraries or kernel classes.

6.1.1 Challenges in refactorings: Two examples of corrupting refactoring

This section details two examples of refactorings that corrupt instances.

6.1.1.1 Pull Up Instance Variable

This refactoring removes the selected instance variable from all the subclasses and defines it in the selected superclass. Figure 6.1 shows the process of applying this refactoring to the *idNumber* instance variable. This instance variable is present in the *Student* and *Teacher* classes. Figure 6.1a shows the original state and Figure 6.1d shows the desired result of the refactoring.

To perform this, the refactoring does the following operations:

1. Iterate all the subclasses of the selected class. If the subclass has the instance variable, the instance variable is removed. Figure 6.1b shows the removal of the *idNumber* instance variable from *Student* and *Teacher* classes.
2. Add the instance variable to the selected class. Figure 6.1c shows the addition of the instance variable *idNumber* to the *Person* class.

During each of the two operations, live instances are migrated due to the change in their structure. This migration is performed by the live programming environment each time a class is modified. During this process, the value of the pulled up instance variable is lost for live instances of subclasses (*e.g.*, Student, Teacher). These values are lost because the instance variables from subclasses are removed during the first migration step. Note that the order of these steps cannot be changed because instance variables of subclasses should be removed before adding the new instance variable to the superclass to avoid duplicated instance variables.

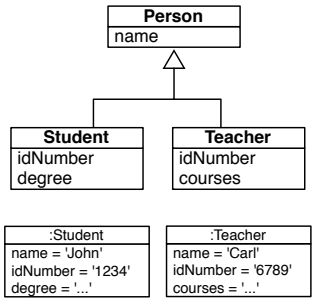
When we compare the result of applying the *Step 2* (Figure 6.1c) and the expected result (Figure 6.1d), we can see that the refactoring is not preserving live instance state. A state preserving refactoring must correctly keep the state of *idNumber* in existing instances.

6.1.1.2 Split Class Refactoring

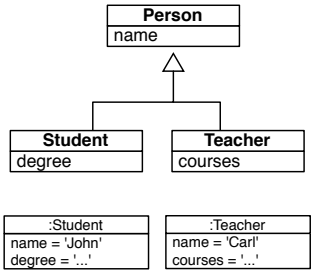
This refactoring extracts a selected subset of instance variables into a new object. It replaces all accesses to the selected instance variables by message sends to the new object. It also present the same problem described before.

This refactoring is more complex than the previous one. Indeed, to perform this refactoring the following changes are performed:

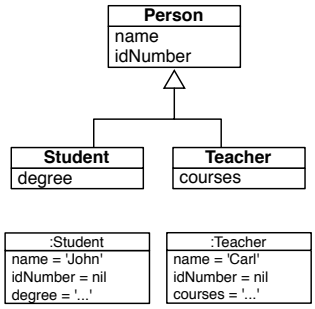
1. Create a new class with the selected instance variables.



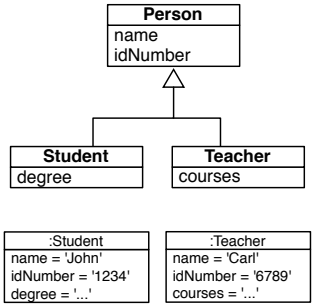
(a) Original State before the refactoring



(b) Step 1: Remove subclass instance variables



(c) Step 2: Add superclass instance variable



(d) Expected Result

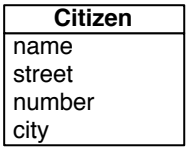
Figure 6.1: Step by Step of appliying the Pull Up Instance Variable refactoring to the *idNumber* instance variable present in *Student* and *Teacher* classes.

2. Add the accessor methods to the new class.
3. Add a new instance variable in the original class to hold the extracted object.
4. Change all the uses of the selected instance variables with messages to the new object.
5. Remove the selected instance variables from the selected class.
6. Add initialization code creating an instance of the new class when the selected objects are created.

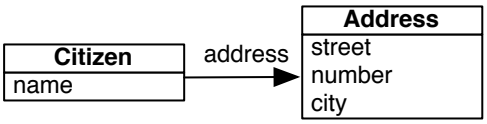
Figure 6.2a depicts the class structure of an example and Figure 6.2c shows some live instances in the environment before applying the refactoring. As a contrast, Figure 6.2b shows the expected result of applying the refactoring with the desired state of the live instances in Figure 6.2e.

Even though, the class structure and the methods are correctly created, live instance state is not preserved. Figure 6.2d shows the actual result of applying this refactoring. Since there is no special handling for migrating the extracted instance variables, this state is lost.

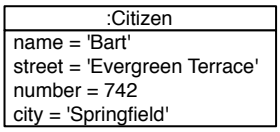
Although the refactoring operation is able to perform all the structural



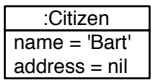
(a) Classes before Refactoring



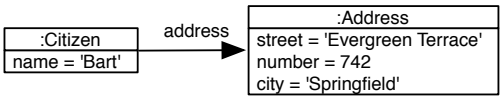
(b) Classes after Refactoring



(c) Objects before Refactoring



(d) Actual Result



(e) Expected Result

Figure 6.2: The Split Class refactoring corrupts its instances.

and behavioural changes needed, the instances are not migrated properly. The instances of the selected class are now useless because all the selected instance variables have been removed, replaced by an empty instance variable, and all the code has been modified to use this empty instance variable.

6.1.2 Refactoring Impact Categories

Instance corruption is not only present in the described examples. Instance corruption exists in a larger set of refactorings. Considering the 72 refactorings described in Fowler's book *Refactoring: Improving the Design of Existing Code* [Fow99] as a set of existing refactorings, we analyse the impact of applying the refactorings over live instances. This analysis shows that 36.11 % of these refactorings produce *instance corruption* when applied in presence of live instances. Preventing instance corruption is not just a matter of adding new pre/post-conditions to refactorings. Indeed, refactorings have pre and/or post-conditions as part of their definition. These conditions help to guarantee consistency. Nevertheless, in the literature, these conditions only focus on structure and behaviour consistency without taking care of instances. Extending pre/post-conditions is not enough because instances must be correctly migrated according to the applied refactoring and the

context.

We classified refactorings into 4 different categories related to *instance corruption*. For each category, we assess the amount of work to be able to preserve instance state.

No Corruption. The live instances are not affected at all because the refactoring does not modify the structure or the use of the state. All the changes are in the methods of the object. An example of this category is *Add Parameter refactoring*. This refactoring only adds a new parameter to an existing method. The method is modified in the class, but the structure of the live instances is not modified and no migration is required.

Internal Corruption. The structure of live instances is modified, but the state preservation can be computed using exclusively the modified instances. Client objects that have references to the modified instances don't need to be updated. An example of this is the *Instance variable rename* refactoring, where the value of the renamed instance variable should be preserved in a new instance variable with a new name. Another example is the *Extract class* refactoring. Here, the value of one or more instance variables is replaced with an object but the information to create this new object is taken from the original instance.

Class Corruption. When a refactoring changes the class of a set of instances, some or all selected instances should be migrated because their structure may also have changed which requires a data transformation. The *Introduce Local Extension* refactoring is an example of this kind of instance corruption. This refactoring moves part of the behavior of selected class to a new class that contains these extra methods. The new class is a subclass of the selected class. Some or all of the instances of the selected class should be migrated to instances of the new class.

Complex Corruption. The changes performed by the refactoring either impact or require access to more instances than the ones from the modified classes to keep consistency. One example of refactoring corrupting more instances than the ones from the modified classes is *Change Value to Reference*. This refactoring impacts not only the instances becoming references but also all the users of those instances. Think about two clients referencing two equal value objects that when converted to references should be the same instance. This refactoring impacts both the client and the transformed value objects. On the other hand, an example of a refactoring that requires access to many instances is *Change Unidirectional Association to Bidirectional*. The refactoring

creates a bidirectional association from a unidirectional. For example consider a *Course* with a collection of *Student*. This refactoring requires access to *Course* instances to insert the back pointer to *Student* instances.

In Appendix C we present the detailed classification and the justification of each of the problematic refactorings.

Applying this classification we discovered that 26 out of 72 (36.11%) refactorings corrupt instances and should take care of the migration of live instances to conserve their integrity. This means that 36.11% of these refactorings cannot be applied in a live programming environment without *corrupting* instances in the running program.

6.1.3 Ubiquity of the problem

This analysis is not exclusive to the Fowler's catalog and we also extended it to existing industrial tools. We wanted to check if potential problems are present in tools used daily by every programmer. To show this we extend the analysis to refactorings tools in *Rewrite Engine for Pharo (Smalltalk)*, *Eclipse JDT (Java)*, *Groovy/Grails Tool Suite (Groovy)*, *IntelliJ IDEA (Java)*, *Visual Studio 2015 with ReSharper (C#)* and *WebStorm (Javascript)*.

Table 6.1 presents the results of the tool analysis and how they compare with the results of Fowler's list of refactorings. These results show that if the refactoring tools are used in live programming environments, the consistency of the live instances is not preserved.

The analysed industrial tools are used in live programming environments. When doing so, they present the described problems. For example, live programming is performed in Java or Groovy using tools like DCEVM [WWS10], JRebel [Zer12] or Jvolve [SHM09].

Also, studies of the usage of automatic refactorings show that the problematic refactorings are used daily [VCN+12,NCV+13,MHPB12]. Other studies show that the use of automatic refactorings is limited as it is not giving enough guarantee to developers [KZN12]. Although there are no specific studies applied to live programming, it is possible to extrapolate these results to live programming.

6.2 Our Solution: Atomic Refactorings for Live Programming

Traditionally, refactorings are applied in a *non-atomic way* and changes generated by the refactoring tool are applied one at a time, modifying live instances after each change. For example, the rename refactoring involves two operations: adding a new instance variable with the new name and removing the

	No Corruption	**Internal Corruption**	**Complex Corruption**	**Class Corruption**	**Total Corruption**
Fowler	46 (63.89%)	9 (12.50%)	11 (15.28%)	6 (8.33%)	26 (36.11%)
Eclipse JDT	24 (72.73%)	5 (15.15%)	3 (9.09%)	1 (3.03%)	9 (27.27%)
Resharper	32 (69.57%)	9 (19.57%)	4 (8.70%)	1 (2.17%)	14 (30.43%)
IntelliJ IDEA	28 (66.67%)	5 (11.90%)	8 (19.05%)	1 (2.38%)	14 (33.33%)
Pharo	33 (66.00%)	13 (26.00%)	2 (4.00%)	2 (4.00%)	17 (34.00%)
WebStorm	10 (90.91%)	1 (9.09%)	0 (0.00%)	0 (0.00%)	1 (9.09%)
Groovy	7 (63.64%)	3 (27.27%)	0 (0.00%)	1 (9.09%)	4 (36.36%)
Average	**25.71 (67.92%)**	**6.43 (16.98%)**	**4.00 (10.57%)**	**1.71 (4.53%)**	**12.1 (32.08%)**

Table 6.1: Results of the analysis of existing refactoring engines.

instance variable with the old name. These operations are performed one after the other and after each one, live instances are migrated and corrupted.

In a live programming environment, atomic refactorings are needed to prevent *instance corruption*. Since instances are accessed and modified concurrently in a multi-threading application, all the changes should be applied at once guaranteeing isolation and mutual-exclusion.

Next, we present our solution to perform refactorings that preserve the state of live instances. This solution is based on a Dynamic Software Update mechanism. It manages the application of changes to methods and class structure as well as the migration of existing instances from an old class structure to a new one. The DSU mechanism performs all the changes atomically, preserving the state of the application and its behaviour while the application is running. For this solution we implemented the refactoring engine using *gDSU*.

Regarding refactorings, it means that all classes are modified at once and

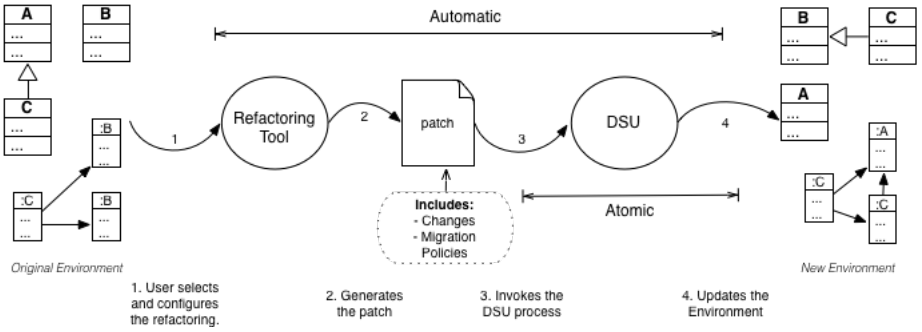


Figure 6.3: The Atomic Refactoring process.

that live instances are directly migrated from the current version to the final one in an atomic way.

Description of the Atomic Refactoring Application

gDSU provides ways of expressing the changes to be applied as a patch. The atomic refactoring engine generates patches automatically. As the changes required by an automatic refactoring are well known, the patch is generated with this changes. Instead of applying the changes in the methods and classes they are accumulated in a patch. All the changes are scheduled and performed atomically, *gDSU* determines the exact moment to execute the update.

Figure 6.3 describes the overall process, the following section expands the details:

- In *Step 1*, the user selects and configures the refactoring, this is performed through the same user interface the user uses in non-atomic refactorings. When applying a refactoring, our solution relies on the same information that is provided in the non-atomic implementation of the refactorings. Indeed, all required information is already specified in the refactoring to apply or it is retrieved from the live environment. After having all the user input, the process is fully automatic.
- In *Step 2*, the atomic refactoring engine generates the *patch*. As described before, this patch includes all the modifications needed and the migration policies to migrate live instances. The creation of the patch is specified in the refactoring definition. Most of the time, existing refactoring engines compute the changes to be applied to classes and methods. With Atomic Refactoring, refactorings should also compute the migration policies that we will be applied to the instances.
- The *patch* is the sole entry parameter of *gDSU*. In *Step 3*, *gDSU* is invoked to apply the patch.

- *In Step 4*, *gDSU* applies all the changes in an atomic way. Preserving the behavioural consistency [RBJO96] and also the consistency of live instances. *gDSU* can be executed while the system is running. *gDSU* selects the best moment to execute the update process, checking that none of the running threads is using the affected instances. As the changes are applied atomically, in case of an error or problem during the execution the process is safely aborted.

After an update of the running environment by *gDSU*, the atomic refactoring is completed and the user can continue using it.

6.3 Preserving Instance State when Applying Refactorings with gDSU

Our refactoring model and *gDSU* are successfully used to correctly handle live instances when applying refactorings presented in Section 6.1. In essence, our refactoring engine generates the corresponding migration policy for existing instances according to the currently applied refactoring. We have applied this technique to all the refactorings in Pharo[1] that corrupt instances.

6.3.1 Pull Up Instance Variable

In the case of *Pull Up Instance Variable* the migration policy should correctly copy the values from the instances before the application of the refactoring to the refactored instances. There is no need of transforming the values, just not losing them and correctly re-assign them into the refactored instances. To achieve this, the atomic refactoring engine generates a migration policy for the refactored class and its subclasses. This policy copies the state of all affected instances into newly created refactored instances. The *gDSU* creates the refactored instances in the new environment. The refactoring engine correctly initializes new instances with the saved state even if refactored objects have a different layout from original ones (e.g. instance variable order has changed) by using the instance variable names. Relying on names is the default migration policy and it is automatically generated. Here the automatically generated code for the migration policy of a pull up refactoring:

```
migrateInstance: new fromOldInstance: old inNewEnv: newEnv fromOldEnv: oldEnv
    new class instanceVariables do: [ :newIV |
      old class instanceVariableNamed: newIV name
        ifFound: [ :oldIV | newIV write: (oldIV read: old) to: new ] ].
```

[1] Atomic refactoring implementation is available at
https://github.com/tesonep/pharo-atomic-refactors

This migration policies iterates all the instance variables defined in the new instance. In Smalltalk, the instance variables are reificated as objects. They can be accessed by the class of an object. The code iterates the instance variables of the new instance, and looks for the one with the new name in the old instance. If the instance variable is found in both instances, the object representing the instance variable is used to read it from the old instance and to write it in the new instance.

6.3.2 Split Class Refactoring

For solving the live instance migration of Split Class, the atomic refactoring engine generates a more complex migration policy. Although this migration policy is more complex because it has transformations of values, the migration policy is generated automatically.

We call mother objects the instances of the selected class for the refactoring and child objects the instances of the newly created class. We have two versions of the mother object, the old and the new one. This is shown in the Figure 6.2c and Figure 6.2e. The first figure shows the instance before the migration, with all the values of the instance even the ones to be migrated to the new class. And the second figure shows the two instances that are the desired result of the refactoring. This migration policy performs the following steps for each of the instances to migrate:

1. Create a new instance of the child class.
2. Store this new object into the instance variable of the new mother object.
3. Copy all the instance variables to extract to the new child object.

The creation of the new instances, the copy of all the instance variable values to the new created instance and building the relationship between the migrated mother object and the new child object are not performed by default by *gDSU*. It is the responsibility of the *migration policy* to perform the following tasks:

- It creates a new child instance.
- From the old mother instance, it copies all the instance variable values to the newly created object.
- It adds a reference to the child object in the new mother instance. Using for this the new instance variable in the mother object.

This migration policy can be automatically generated by the atomic refactoring engine using the information it already has. A detailed explanation of the implementation of this refactoring is presented in Section 6.4.

6.4 Using *gDSU* to preserve instance state

We validate our proposed solution by the implementation of an extension to the refactoring tool present in the *Pharo Programming Environment*. We selected Pharo because (1) it supports live programming with an advanced debugger allowing to define methods on the fly, dynamic recompilation of classes, and other features supporting live programming such as instance migration, (2) the Refactoring engine available in Pharo is the direct evolution of the original Refactoring Browser [RBJ97] implemented in Smalltalk, (3) the Pharo Refactoring engine propose one of the most complete refactoring implementation with 50 refactorings.

We implemented our Atomic Refactoring solution based on the use of *gDSU*. This DSU solution has some characteristics that ease the implementation of the atomic refactorings. It supports the definition of patches. Each patch contains the set of changes, migration policies and validations. This DSU runs as a library without needing to modify the running Virtual Machine or expecting the target code to follow any guideline.

As required by our solution *gDSU* receives a patch describing the modifications. The patch contains not only all the changes to perform but also the migration policies to apply. *gDSU* allows us to perform all the changes in an atomic way, which means all the modifications are applied at once.

Also, it allows the use of migration policies to describe how the objects should be migrated from the old version to the new version. *gDSU* allows an easy reuse of the migration policies. This feature eases the development of the migration policies for the different refactorings. Moreover, *gDSU* provides a number of default migration policies that can be used. However, in our implementation, we need to implement our own migration policies to be able to express more complex instance migrations.

gDSU also allows us to validate the correct execution of the update. The validations objects are added to the patch and they are generated by the refactoring tool. The validations are defined for each of the automatic refactorings. Each automatic refactoring includes the validations as it includes the required operations to do. For example, in the *Pull-Up* refactoring instances of the new classes are validated to have the proper instance variable value.

Using them we validate the correctness of the refactoring after all the changes are applied. Again *gDSU* allows the reuse of different validations in different refactorings. This feature is used when a refactoring operation needs to validate a post-execution condition. Including validations is not mandatory, but provides a way of validating the result of the refactoring execution, not only over the static model but also on live instances. For example, it can be used in the *Split Class Refactoring* to validate if the accessor methods in the

original instances return the same values.

Our solution is handling each refactoring as an atomic operation. When a set of two or more refactorings modify the same classes, the required migration policy should be capable of performing the migration taking into account both refactorings. The required migration gets more and more complex. This required complexity is outside the scope of our solution.

As our solution is generating the migration policies and validations automatically based in the definition of the refactorings. We implemented for each refactoring the code to generate them. This automation constraints our solution to apply each refactor atomically. We does not support to apply two or more refactorings at the same time. As they might modify the same class in many ways that cannot be analysed by the automatic generation of the migration policies.

We decided to implement the refactoring tool as an extension, so we can reuse all the user interface and the integration of existent tool with the IDE. As the required information to apply a refactoring is the same required by the previous implementation of the refactoring tool, it was not needed to implement modifications in the user interface. The implementation is available in Github[2] and can be easily downloaded and used in Pharo 6.

6.5 Application of the Refactoring step by step

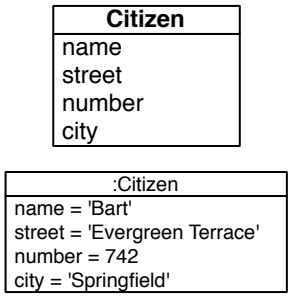


Figure 6.4: State before refactoring

Our proposed solution performs a number of steps to apply refactorings in an atomic fashion. We will explain in detail these steps following the example presented in Subsection 6.3.2.

Figure 6.4 shows the state before applying the changes. To perform the changes the atomic refactorings apply the following steps.

1. The *new environment* is created. All the new and modified objects and class/methods will be stored in this new environment. The environment starts empty. (Figure 6.5)

[2]https://github.com/tesonep/pharo-atomic-refactors

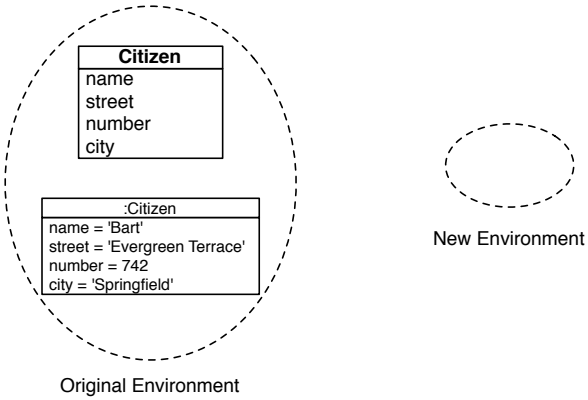


Figure 6.5: A new environment is created

2. The modified versions of the classes *Citizen* and *Address* are created in the *new environment*. The refactoring operations modify the copied classes without tampering the *original environment*. For example, it is not needed to create the class at once, as the *Address* class can be created empty and then add the different instance variables (Figure 6.6).

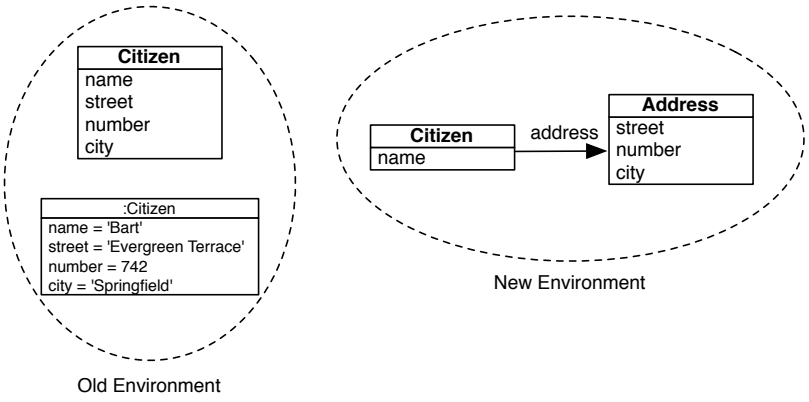


Figure 6.6: All the modifications to the classes are applied

3. Once the class modifications are completed, the *Citizen* instances should be migrated. The migration policy migrates the *Citizen* instances, creating the *Address* instance needed in each case (Figure 6.7). This migration is performed using the refactoring migration policy. This policy is included in the automatic refactoring engine, and it is used only for this refactoring. The following code is the migration policy.

```
SplitClassMigrator >> migrateInstance: new fromOldInstance: old inNewEnv: newEnv
      fromOldEnv: oldEnv
   | child childClass targetClass oldClass oldVariable |

  "Step 1:uses the default migration for the mother object, that
   copy all the instance variables by name.
   This is implemented in the reusable part of the migration policies.
   It copies the instance variable values by name, copying the instance
   variables existing in the old instances into the new instance."
   self basicMigrateInstance: new from: old.

   "Step 2: recovers the newly child class created from the environment.
   The newClassName is a parameter of the refactoring."
   childClass := newEnv at: newClassName.

   "Step 3: creates a new instance of the child class.
   Using the class recovered from the environment the new instance is created."
   child := childClass new.
   targetClass := new class.
   oldClass := old class.

   "Step 4: add the reference from the mother object to the child object."
   (targetClass instanceVariableNamed: referenceVariableName)
      write: child to: new.


   "Step 5: copy all the extracted instance variables.
  Fill up the child instances with the variables extracted from the mother instance.
   These are the instance variables that are extracted from the mother class.
   These variables are a parameter to the automatic refactoring."

   variablesNamesToExtract
      do: [ :e |
          oldVariable := oldClass instanceVariableNamed: e.

         (childClass instanceVariableNamed: e)
              write: (oldVariable read: old) to: child ].
```

4. When all live instances are migrated, the update engine validates the correctness of the refactoring, by the execution of a validation object generated by the refactoring tool. In this case, it checks that all the *Citizen* instances have a corresponding *Address*. As the validation is successful, the modified objects in the old environment are replaced with the ones in the new environment. The bulk swap operation replaces all the instances updating the references to them. (Figure 6.8)

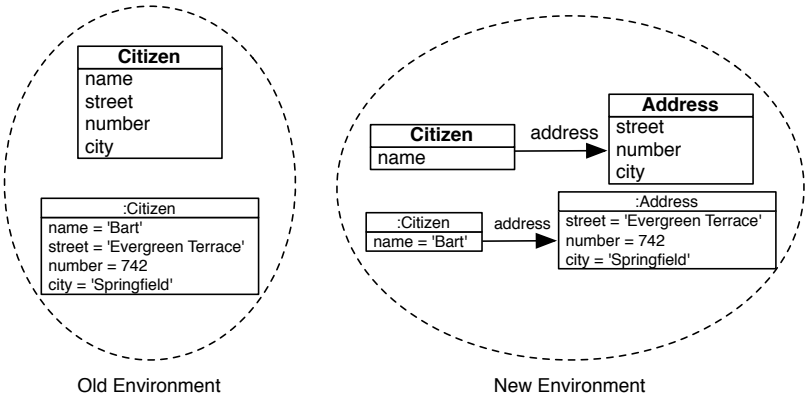


Figure 6.7: Live instances are migrated

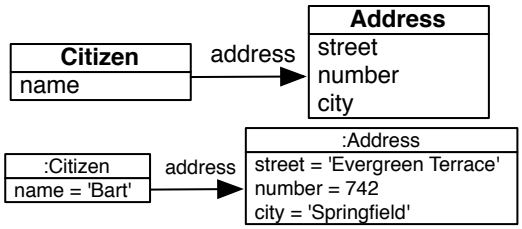


Figure 6.8: The *New environment* replaces the old environment

6.6 Validation

This section presents three tested scenarios to validate the correct application of the changes preserving the state of the application. The validations have been done using the implementation that is publicly available in Github[3]. In such repository there are instructions to install our solution in Pharo 6.

6.6.1 Validation 1: Refactorings without Corruption

Research Question. Is it possible to execute refactorings that do not produce instance corruption?

Scenario. This validation shows that our solution does not affect the existing refactorings. For this test we use the Protect Variable Refactoring (implemented by RBProtectVariableRefactoring). This refactoring removes the accessors to an instance variable and replaces all the uses of these messages in the class and subclasses with direct accesses to the instance variable.

Figure 6.9 shows the original state before the execution of the refactoring. Figure 6.10 shows the state after the execution of the refactoring.

For testing the migration of instances, we have created instances of the class Person with state and validated that this state is correctly created. Finally,

[3]https://github.com/tesonep/pharo-atomic-refactors

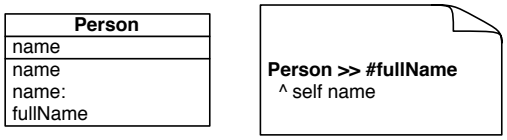


Figure 6.9: Before applying the protect variable refactoring.

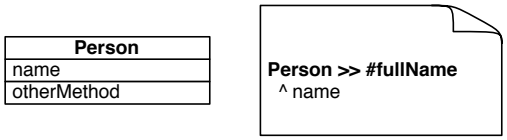


Figure 6.10: After applying the protect variable refactoring.

we execute the refactoring programmatically. Listing 6.1 shows the executed code.

```
refactor := RBProtectVariableRefactoring
  instanceVariable: 'anInstanceVariable'
  class: classProtectVariable name.
refactor execute.
```

Listing 6.1: Execution of the Protect Variable Refactoring

Result. After executing the refactoring, we validated that the static modifications are performed correctly and the migration of instances has been done correctly.

6.6.2 Validation 2: Refactorings with Internal Corruption

Research Question. Is it possible to execute refactorings that produce internal corruption preserving the state of the application?

Scenario. This validation shows that our solution does not affect the existing refactorings. For this test we use the Pull Up Variable Refactoring (implemented by ARPullUpInstanceVariableRefactoring). This refactoring removes the instance variable from the subclasses and define it in the superclass.

Figure 6.11 shows the original state before the execution of the refactoring. Figure 6.12 shows the state after the execution of the refactoring.

For testing the migration of instances, we have created instances of the class Employee with state and validated that this state is correctly created. Finally, we execute the refactoring programmatically. Listing 6.2 shows the executed code.

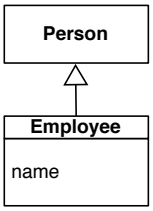


Figure 6.11: Before applying the Pull Up variable refactoring.

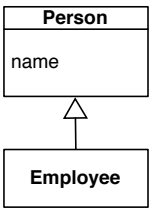


Figure 6.12: After applying the Pull Up variable refactoring.

```
refactor := ARPullUpInstanceVariableRefactoring
  variable: 'name'
  class: Person.
refactor execute.
```

Listing 6.2: Execution of the Pull Up Variable Refactoring

Result. After executing the refactoring, we validated that the static modifications are performed correctly and the migration of instances has been done correctly.

6.6.3 Validation 3: Refactorings with Complex Corruption

Research Question. Is it possible to execute refactorings that produce complex corruption preserving the state of the application?

Scenario. This validation shows that our solution does not affect the existing refactorings. For this test we use the Split Class Refactoring (implemented by ARSplitClassRefactoring). This refactoring extracts a set of instance variables to a new class. The developer chooses which instance variables should be moved into the new class. The new class becomes an instance variable of the original class and every reference to the moved variables is replaced by a accessor call.

Figure 6.13 shows the original state before the execution of the refactoring. Figure 6.14 shows the state after the execution of the refactoring.

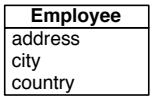


Figure 6.13: Before applying the Split Class refactoring.

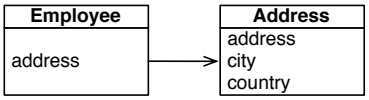


Figure 6.14: After applying the Split Class refactoring.

For testing the migration of instances, we have created instances of the class Employee with state and validated that this state is correctly created. Finally, we execute the refactoring programmatically. Listing 6.3 shows the executed code.

```
refactor := ARSplitClassRefactoring
  class: Employee
  instanceVariables: #(address city country)
  newClassName: #Address
  referenceVariableName: #address.
refactor execute.
```

Listing 6.3: Execution of the Split Class Refactoring

Result. After executing the refactoring, we validated that the static modifications are performed correctly and the migration of instances has been done correctly.

6.7 Conclusion

In this chapter, we identify the problem of refactorings corrupting instances when there are live instances of the modified classes. We propose a categorization of the *instance corruption* issue. And we show that 36.11% of refactorings described in the literature present this problem. Moreover, we proposed a refactoring implementation mechanism which solves this problem.

Our Atomic Refactorings mechanism preserves instances' state. It is suitable for live programming environments because it does not corrupt instances of refactored classes. We use *gDSU*, it offers the possibility of modifying all the objects in a separated environment. When all the changes are performed, it replaces all the modified objects at once.

We presented a validation of our proposed solution with an implementation of an atomic automatic refactoring tool in Pharo. This validation shows

that the proposed solution cover the stated requirements for a set of automatic refactorings that does not produce instance corruption. So, this solution is applicable to live programming environments without affecting the stability of the running application.

Our solution is integrated with a refactoring tool existing in an industrial platform. The use of this new extension is transparent to the developer. Developers perform the refactorings without thinking to regenerate the live instances.

State-aware Transactional Live Programming

Chapter 7

Contents

Live programming environments [San78], such as Pharo [BDN+09], allow developers to modify the running application while it is executing. These environments include not only the code of the application but all the live instances representing the application state.

As stated before, live programming environments implement simple yet powerful mechanisms to migrate the application state after each change. Pharo allows the developers to freely modify application code, core libraries and the language itself. It allows us to modify all the elements present in the environment.

However, this mechanism does not cover all the possible changes leaving uninitialized variables and inconsistent global state. As a result, some modifications affect the stability of the system. Complex modifications require special care from the developer to maintain the stability of the running application (*i.e.*, staging, sequencing, doing intermediate changes). As this special care is not always possible [PDF+15], this requirement limits the flexibility and power of live programming environments. Moreover, even a modification in application code could arise a stability problem.

We propose to address this limitation scoping the modifications to a transactional environment. By doing so, the modifications to classes, methods and instances do not affect the running application. Scoping the changes is not enough, the developer should be able to perform new changes, test her changes and inspect and modify live instances.

Also, the developer should be able to safely discard or apply these changes. As the application is running, the application of changes should be performed atomically. The changes to be applied are not only changes to methods or classes, also live instances should be migrated to maintain the application state coherence.

We implemented our solution in Pharo as an extension of *gDSU*. Our prototype tool (*PTm*) allows the developer to evaluate changes in a scoped environment. This environment includes the unmodified classes and instances from the application environment and the modified ones. This environment is created in an efficient way using the techniques included in *gDSU*, only containing the referenced or modified classes and instances. This alternative environment is used to evaluate expressions, inspect and modify the instances and classes, allowing the developer to test her modifications. Once the modifications are ready, the developer applies or discard them.

In case of needing migration strategies, the environment provides ways of detecting the need and provide ways of configuring the required migration strategies.

There are other solutions that provide a scoped transactional environment to isolate the changes [DGL+07,LH12,WLN13,PS87,MRH17,CPDD09]. However, these solutions do not take into account the migration of application state.

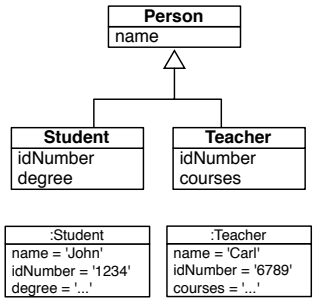
7.1 Changes Corrupting Instances

Needing Transactions. Pharo allows the user to modify the class structures and the methods of a running application. However, to maintain the coherence of the application state these changes have to be performed atomically or in a sequence of well-designed steps. Also, there are situations where such sequencing is not possible [PDF+15] requiring the atomic application of the changes.

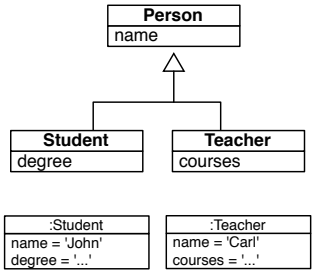
An example of a complex change is the *Pull up* refactoring (Figure 7.1) that we introduced in Chapter 6. The refactorings are a good example of complex changes, where one or more classes and methods are modified in the same operation. For this example, the developer will perform the changes to apply the refactoring manually. The developer performs the following steps:

- Removes the instance variable idNumber from all the subclasses of Person.
- Adds the instance variable idNumber to the Person class.

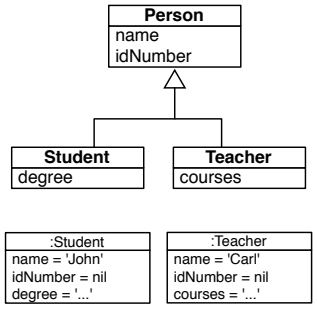
This set of changes produced the expected result in the class structure. However, as stated before if these changes are performed naïvely the state



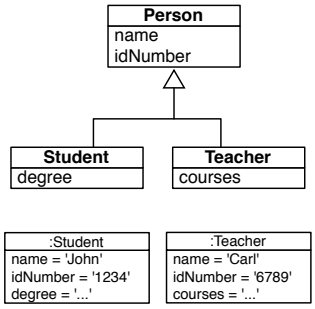
(a) Original State before the refactoring



(b) Step 1: Remove subclass instance variables



(c) Step 2: Add superclass instance variable



(d) Expected Result

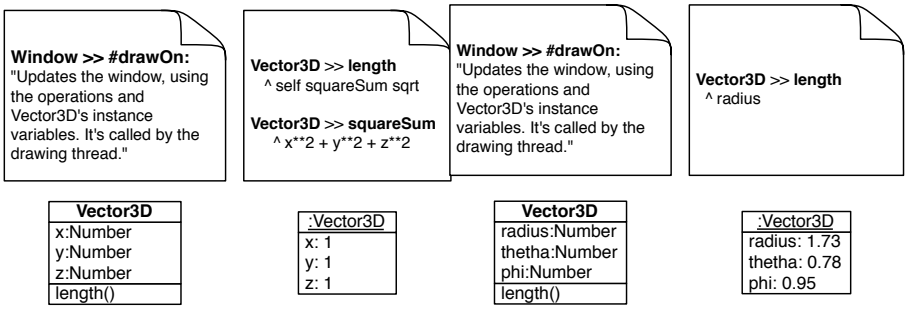
Figure 7.1: Step by Step of applying the Pull Up Instance Variable refactoring to the *idNumber* instance variable present in *Student* and *Teacher* classes.

of live instances is lost. In Pharo (and in many Smalltalks), when the first step is performed the instance variable is removed from all live instances of Student and Teacher. Once the instance variable is removed, their values are lost. When the instance variable is added back to the class Person, the instance variable values cannot be restored as they are already lost. This modification requires the atomic application of the changes and performing the instance migration after all the changes in the classes are performed.

Needing migrations. There are changes that require more than transactional changes, they require correct migration of instance state. If the modifications affect the structure of live instances or the instance variable value types, live instance should be migrated using custom logic [TPF+16].

In Figure 7.2, we return to the same example introduced before. It is a simple application that draws vectors in a window. The vectors are represented by instances of Vector3D. These instances have cartesian coordinates (x, y, z).

Figure 7.2a shows the initial implementation of the example and Figure 7.2b shows the desired changes to perform. The developer wants to change the representation of the vectors to use polar coordinates. To do so, the following changes are required:



(a) Original state (b) After the desired change

Figure 7.2: Example of changes requiring migration of instances with custom logic.

- Update the methods Window » #drawOn: and Vector3D » #length.
- Remove the method Vector3D » #squareSum.
- Remove the instance variables x, y and z.
- Add the instance variables radius, thetha and phi.

After performing these changes, the instances of Vector3D are in an invalid state. When the old instance variables are removed, their values are discarded. Also, the new instance variables are initialized in nil. With this invalid state, the application using the instances crashes.

To avoid this, the instances of Vector3D should be regenerated (*e.g.*, from a persistent store), or they should be migrated. This set of changes requires a custom migration strategy [TPF+16]. Listing 7.1 shows a possible migration block. The block receives an old instance and a new instance.

```
[ :old :new |
  new radius: old length.
  new thetha: (old z / new radius) arcCos.
  new phi: (old y / old x) arcTan.
]
```

Listing 7.1: Migration Logic required for Vector3D

Needing Testing Environment. To safely perform changes in a live programming environment, the developer requires an environment to test the changes is performing. A transactional live programming environment should keep the interactive development experience.

This interactive experience includes: (1) performing the changes, (2) testing and debugging the changes, (3) modifying the changes or performing new ones. All these operations should be done without affecting the stability of the running application.

We considere a key element of live programming environment the continue flow of interaction between the evolving program an the developer. The

risk of getting an unstable environment or application limits the freedom of the developer and cut down the freedom of evolving the system by small steps.

7.2 Transactional Changes

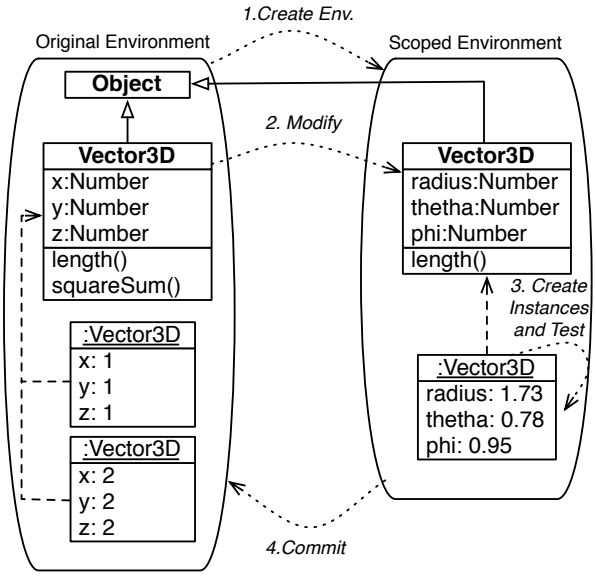


Figure 7.3: Overview of the Solution

We propose a technique to scope changes and instance state modifications in a transactional environment. To do so, we extend the alternative environment proposed by *gDSU*.

This alternative environment is used to evaluate the code modifying the methods, classes and instances. In a nutshell, the developer is working in a copy of the environment and when the changes are complete and safe to apply, the old environment is replaced by the new objects.

Our proposed solution extends the implementation of *gDSU* to be able to execute code and tests in the copied environment. The modifications are applied when they are performed by the user. As all the instances and classes are living objects they are accessible through the environment.

Figure 7.3 shows the overview of the solution and Listing 7.2 presents an overview of how to do the different steps. The integration with the IDE will hide these details.

First, the scoped environment is created empty (Step 1). Next, the developer perform changes to the classes. When the classes are modified, copies of the classes are created in the scoped environment (Step 2). Only the required classes are created.

The developer is free to create instances and test her changes in the scoped environment (Step 3). The classes and instances in the original environment

are not affected.

```
"Step 1: Creation of the environment"
env := TMEnvironment new.

"Step 2: Applying the changes in the scoped environment"
env evaluate: [
    transaction createSubclassOf: #Object withNewName: #Vector3D
      slots: #(radius thetha phi)
      sharedVariables: ''
      package: 'Transactions-Tests'].

"Perform the changes in the methods"
env evaluate: [Vector3D removeSelector: #squareSum].
...
env evaluate: [Vector3D compile: 'phi: aVal. phi:=aVal'].

"Step 3: Create instances and test them"
aVector := env evaluate: [ Vector3DTest new ]
aVector radius: 1.73
...
"Step 4: Commit the transaction"
env evaluate: [transaction commit].
```

Listing 7.2: Using our solution

Finally, when the developer has tested her changes, she commits the operations in the original environment (Step 4).

7.3 Implementing PTm

Our solution extends the implementation of *gDSU*. This extension presents a number of practical issues. We will address these issues in this section. First, we analyse how the new environment is created, which are the elements to include (*e.g.*, classes, objects and methods) in it and when the new environment is created and populated (Section 7.3.1). Second, we provide a solution for state migration from and to the alternative environment (Section 7.3.5). This migration has to handle the globally accessible state (Section 7.3.2). Before applying the changes, it needs to detect and handle the conflicts in the state of the application (Section 7.3.3). Finally, we provide the means to implement the application of changes (Section 7.3.4).

7.3.1 Scoped Environment

An environment includes all the objects, classes and methods required to execute the application. In Pharo, the whole image is the environment of execution. All the modifications performed in our solution are performed in a copy of the environment. This copy is incrementally created. This implementation

follows the same optimizations performed in *gDSU*. However, the discovery of the classes and instances to copy in the new environment is different.

The classes are created in the new environment when they are referenced directly in the expressions evaluated in the environment or referenced in a method that is present in the environment. When a class is referenced in the new environment, the class is copied with the same definition that exists in the original environment. The superclasses of the copied classes are also copied. Also, the methods are compiled in the new environment and stored in the copied class.

To minimize the number of copied classes, we defined a set of classes that are only copied when they are modified (not when they are accessed). This set of classes includes core system Classes (*e.g.*, Object, Array, SmallInteger, Class, Metaclass). Not copying them on access improves the copy in most of the development scenarios, and allowing to copy them when modified allow us to handle transactional changes on them. This improvement is required to make the solution practical, without it the copy of the environment is not practical for speed reason.

To transparently replace all the references to classes inside the alternative environment, the methods and expressions are compiled using the alternative environment to resolve the bindings. When a new binding is required, the environment creates a copy of it and creates the required class. This detection of the bindings and global variables is achieved by plugin on the compiler.

7.3.2 Global State

To be able to execute code in the scoped environment, our solution migrates the required globally accessible state of the application. In Pharo, the global state basically is of two types (1) global variables defined in the environment and (2) class-side variables. All the global state required by the scoped environment is copied in the new environment transparently.

The global variables are copied on access. Whenever an expression or method is compiled using a global variable this global variable is copied to the new scoped environment. We extend the compiler to perform the copy during the compilation of methods. Extending the compiler allowed us to only copy the necessary global values. For example, literals, constants, and shared objects (*e.g.*, true, false, nil) are not copied.

The class-side variables are copied when the classes are included in the scoped environment.

In both cases, the value of the global state is copied. The instances pointed by the global state are copied in the scoped environment allowing us to be

modified freely without affecting the running application.

7.3.3 State Conflicts Detection

If the migration cannot be performed automatically, the transactional environment cannot be committed in the original environment. Any attempt to commit it will raise an exception. To be able to apply the changes in the original environment, the developer should provide the missing migration strategies (Section 7.3.5).

After performing changes in the application code, the structure or use of the instance variables might be altered. If this is the case, these changes require migration of the instances from the old to the new version.

Our proposed solution first detects if the instance migration can be performed automatically. This is performed if one of the following conditions are met:

- There are no live instances (or subclasses instances) of the modified class in the old environment.
- The class structure is not changed in the number of slots or the names of slots.
- The types of the instances in the old environment and new environment are equivalent.
- The global state of the classes is the same (*i.e.*, all their class-side and shared variable values are the same).

7.3.4 Applying Changes

The application of changes is performed following the same guidelines of *gDSU*. The safe point update is calculated in the same way as implemented in *gDSU*.

7.3.5 State-Migration

The migration of state is performed in two stages by our solution. The first stage is the instance migration. In this stage live instances of the old environment are migrated to their new representation. The migration of state is performed using the *gDSU* mechanisms. Our transactional solution calculates the required migration policies to migrate the state of live instances and the global accessible state.

If the changes in the class structure do not add a new slot, the migration is performed automatically. Our solution is able to handle changes in the order of the slots, the position of the slot in the inheritance hierarchy (*i.e.*, moving a

field to a super / sub class) or removing slots. In the automatic migration, the slots are copied from the old instances to the new instances by name. If the migration cannot be performed automatically, the developer should provide a migration strategy.

The second stage is the migration of global state. If the global state has been modified in the scoped environment, the developer should choose between three strategies: (1) using the state in the scoped environment (*using new global state*) or using the state in the original environment (*using old global state*). The selection of the strategy is performed in a class base. The developer selects for each of the classes with modified state which strategy to use.

7.3.6 Aborting the Transaction

To safely abort the transaction, our solution is using the abort mechanism implemented in *gDSU*. We have extended the mechanism to allow discarding the copied environment in any stage before the commit.

Once the new environment is discarded, there is no modifications left in the original instances and classes.

7.4 Using PTm to safely apply changes

To validate our proposed solution, we implemented our tool in Pharo 7[1]. *PTm* provides an alternative environment to execute any modification in a scoped environment. The prototype is available as a Github project[2]. With this prototype, it is possible to perform the changes, execute code and tests, configure the migration strategies and commit or discard the changes.

We show in this section how to update the running application in the examples presented in Section 7.1. As our solution is still a prototype, it is not integrated with the UI or the existing tools. However, it can be used from a workspace (GT Playground). So, in the examples, we will show the code to evaluate. The shown code only includes the required steps to implement the changes, although in the environment and in the development session many expressions and tests could be run.

7.4.1 Transactional Changes

The example of the *Pull-up* refactoring requires transactional applications of the changes. Listing 7.3 shows the steps to perform this update using *PTm*. First, this example redefines both Teacher and Student classes removing the idNumber instance variable. The definition of the classes is performed through

[1] https://pharo.org
[2] https://github.com/tesonep/transactions

the transaction object. This object is available in the context of the new environment. This object is the main entry point of *PTm*.

```
"Creates a new environment"
env := TMEnvironment new.

"Redefines the Student class, removing idNumber"
env evaluate: [
  transaction createSubclassOf: #Person withNewName: #Student
    slots: #(degree)
    sharedVariables: ''
    package: 'Transactions-Example'].

"Redefines the Teacher class, removing idNumber"
env evaluate: [
  transaction createSubclassOf: #Person withNewName: #Teacher
    slots: #(courses)
    sharedVariables: ''
    package: 'Transactions-Example'].

"Redefines the Person class, adding idNumber"
env evaluate: [
  transaction createSubclassOf: #Object withNewName: #Person
    slots: #(name idNumber)
    sharedVariables: ''
    package: 'Transactions-Example'].

"Runs the tests to evaluate that the changes are ok"
result := env evaluate: [ StudentTest suite run ].
result defect isEmpty.
result := env evaluate: [ TeacherTest suite run ].
result defect isEmpty.

"Commits the transaction"
env evaluate: [ transaction commit ].
```

Listing 7.3: Atomic application of *Pull-up* refactoring using *PTm*

Then, the Person class is redefined to have the new instance variable. Finally, the commit is performed. The changes are applied atomically. In this example, there is no need of custom instance migration, as the instance structure of Student or Teacher have not changed. The migration from the old to the new instances is performed automatically by name. This automatic migration is only possible as the changes are applied atomically.

7.4.2 Custom Migration

The second example in Section 7.1, the one migrating the structure of Vector3D class, requires custom migration of live instances. Listing 7.4 shows the

step to safely perform this update using *PTm*.

If the developer tries to commit the transaction before setting a migration strategy and there are live instances of Vector3D, the operation will fail and it will produce an exception informing the situation to the developer. Then, the developer can decide to provide a migration strategy or discard the transaction without affecting the running application.

7.5 Transactional Modifications Validation

This section presents two scenarios tested to validate PTm transactional modification implementation. The validations have been done using the implementation that is publicly available in Github[3].

7.5.1 Validation 1: Manual Refactorings

Research Question. Is it possible to perform transactional manual refactorings preserving the state of the application?

Scenario. This scenario demonstrates that the implementation of transactional modifications supports manual refactorings that require a migration of state even when the changes in the refactoring are applied manually. For this scenario, we use the example of the pull up refactoring, but in this time the refactoring is manually applied. It means the developer changes the classes by hand.

Figure 7.4 shows the original state before the execution of the changes. In this scenario there are two subclasses of the class Person, both having the idNumber instance variable. This instance variable is migrated to the superclass in the changes. Figure 7.5 shows the final state of the application. These changes usually requires caring about the state of the application not to lose state.

To validate it, we create instance variables before executing the changes in the transaction. These instances should be correctly migrated.

Listing 7.3 shows the execution of the described scenario.

Results. After executing the changes and commiting the transactions, we observe that the changes are applied correctly and the instances are correctly migrated without losing application state.

[3]https://github.com/tesonep/transactions

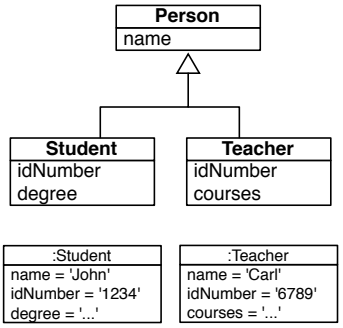


Figure 7.4: State before executing the changes

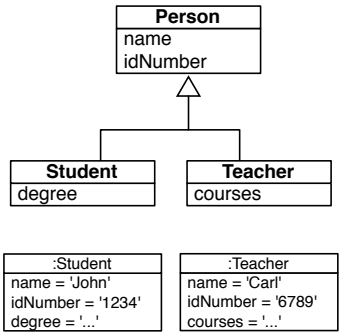


Figure 7.5: State after executing the changes

7.5.2 Validation 2: Detection of Custom Migration Needing

Research Question. When the transaction requires a custom migration, does the proposed solution detect this need and require a custom migration to the developer?

Scenario. For this validation, we use the running example of the modification to Vector3D class. To refresh the example, Figure 7.6 presents the old and new version.

For validating the correctness of the changes, we create instances of Vector3D in the original environment. Later, we execute the code in Listing 7.4. Executing this segment of code produces an error requesting the provision of a migration strategy for Vector3D. After providing the strategy, the transaction is commited correctly.

Results. After executing the changes and commiting the transaction, all the instances of Vector3D are correctly migrated using the provided custom migration.

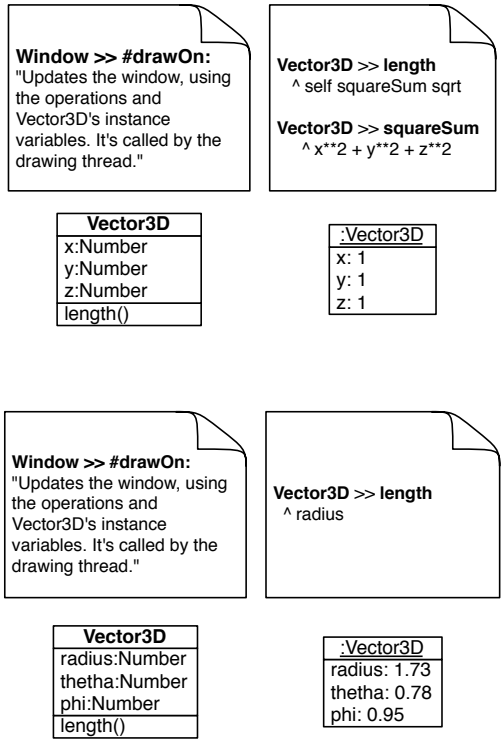


Figure 7.6: Changes requiring a custom migration

7.6 Design Decisions

Nested environments are still not supported by our prototype. Although, one possible implementation may require to implement a polymorphic API between the image environment and the transactional environments.

Block Closures require particular considerations. In Pharo, a closure has a reference to its creating context to be able to access its state. This means that implementation-wise, a shallow copy of the closure will create a new closure sharing the creating context. Thus, any change in the context will affect both closures. In our implementation, the copy of the closures is performed by a shallow copy to minimize the impact in performance. However, this is a limited case as it only applies to globally shared non-clean closures.

Our prototype does not implement any given support for concurrent transactions, the transactions are applied in the order they are committed. However, the detection of conflicts notifies the problems when trying to commit the second transaction. Having a proper locking of elements or versioning is a possible extension to this prototype.

We decided to use blocks to express the migration strategies. Using blocks allows the developer to easily provide a migration strategy. However, it lacks the ability to reuse this migration strategies. We consider to extend the support to using objects as migration strategies allowing a greater reuse.

```
env := TMEnvironment new.
"Redefines the class with the new structure"
env evaluate: [
    transaction createSubclassOf: #Object withNewName: #Vector3D
      slots: #(radius thetha phi)
      sharedVariables: ''
      package: 'Transactions-Tests'].

"Perform the changes in the methods"
env evaluate: [Vector3D removeSelector: #squareSum].
env evaluate: [Vector3D compile: 'length ^ radius'].

env evaluate: [Vector3D removeSelector: #x].
env evaluate: [Vector3D removeSelector: #y].
env evaluate: [Vector3D removeSelector: #z].
env evaluate: [Vector3D removeSelector: #x:].
env evaluate: [Vector3D removeSelector: #y:].
env evaluate: [Vector3D removeSelector: #z:].

env evaluate: [Vector3D compile: 'radius ^ radius'].
env evaluate: [Vector3D compile: 'thetha ^ thetha'].
env evaluate: [Vector3D compile: 'phi ^ phi'].
env evaluate: [Vector3D compile: 'radius: aVal. radius:=aVal'].
env evaluate: [Vector3D compile: 'thetha: aVal. thetha:=aVal'].
env evaluate: [Vector3D compile: 'phi: aVal. phi:=aVal'].

"Run the tests"
results := env evaluate: [ Vector3DTest suite run ]
"Check if the tests are ok"
results defects isEmpty.

"Try to commit the transaction.
It fails informing that a migration
is required for Vector3D"
env evaluate: [transaction commit].
"Provides a migration strategy for Vector3D"
env evaluate: [transaction migrate: Vector3D with: [:old :new |
    new radius: old length.
    new thetha: (old z / new radius) arcCos.
    new phi: (old y / old x) arcTan.
  ]].
"Commits the transaction"
env evaluate: [transaction commit].
```

Listing 7.4: Atomic update of Vector3D using *PTm*

Regarding the migration strategies, we also consider the automatic generation of migration strategies using the information in the image environment and the alternative environment.

Other possible extension is the integration of our transactional environment with the existent tools in the Pharo Image providing a transparent experience to the user.

Finally, a possible enhance to our prototype is the use of Write Barriers and lazy binding of the associations to minimize the number of objects copied in the creation of the new environment.

7.7 Requirements Assessment

This section compares the stated requirements in Chapter 2 with the capabilities of *gDSU* including the extensions described in Chapters 6 and in this chapter. Covering all the requirements makes *gDSU* a practical DSU solution addressing the instance corruption problems and preserving the state of the application. We will center in the requirements that are needed for a live programming DSU.

Isolation. *gDSU* with the transactional extension allows the developer to safely perform her changes, to test those changes, to apply or discard the sandboxed environment. The isolation is guaranteed by the use of a copied environment. *gDSU* provides the required level of isolation.

State Migration. The state migration requirement is covered by *gDSU*. Our proposed solution allows the patch to include the migration logic to express the migration of live instances. This requirement is already covered by the basic implementation of *gDSU*.

Patch Generation. *gDSU* has been extended to collect the changes from the changes in the sandboxed environment. Also, it identifies the changes performed through the use of automatic refactorings. Moreover, *gDSU* propose automatic migration strategies for the common changes in the enviornment, including the automatic refactoring. For the migrations that require business logic, *gDSU* detects this situation and collaborates with the developer to provide a correct migration policy.

Atomicity. The commit operation implemented in *gDSU* guarantees the atomicity of the solution. The implementation correctly modifies the classes and migrates live instances from the original version to the new one.

Automatic Safe Point Detection. *gDSU* implements a safe update point detection algorithm. The implementation of the algorithm does not change for

live programming environments. The conditions to find a safe update point are the same that stated for the Production DSU.

7.8 Conclusion

In this chapter, we center the analysis in the state migration requirements when applying changes to a live programming environment. These problems arise daily when modifying applications with live instances. Based on these problems, we propose a transactional modification solution that allows us to handle these daily problems. The proposed solution is based on *gDSU*. *gDSU* is a DSU tool that is designed to allow us to update programs in live programming environments.

We showed that our solution and prototype are able to handle these problems by applying the changes in a scoped environment and later applying them back to the original environment.

Our tool still requires additional work to be integrated in the existing Pharo tools. This is an important step to allow the developers to transparently benefit from our solution.

From the validation of the requirements, we observe that *gDSU* covers the requirements to provide a safe live programming experience. Comparing *gDSU* with existing solutions shows that our proposed solution covers all the requirements extending the capabilities of live programming environments with the DSU solution safety.

Regarding the automatic generation of patches, we see that the comparison with other solutions might show that our solution is lacking an automatic patch generation. However, the lacking of automatic patch generation is related to the state migration logic. Our solution, in comparison with the other solutions, offers a way of specifying state migration logic for cases where the automatic generation is not enough (*e.g.*, modifications related with the use of an instance variable or changes in the business logic).

Table 7.1 presents the results of comparing our solution with development oriented DSU solutions and classical live programming solutions.

Category	**Requirements**									
	Atomicity	State Migration	Automatic Safe Point Detection	Small Run-time Penalty	Minimal Downtime	Isolation	Patch Generation	Patch Reuse	Self and Core Update	**Examples**
Classical Live Programming	○	◐	○	●	●	○	A	○	◐	Lisp, Clos, Smalltalk
Development DSUs	●	○	○	○	◐	◐	A	○	○	Jrebel, Javeleon, Jvolve
gDSU	●	●	●	●	●	●	S	●	●	

●: Yes ○: No ◐: Limited
A: Automatic M: Manual S: Semi-Automatic

Table 7.1: gDSU vs. Development DSU & Live Programming Environments

Part IV

Conclusion

Chapter 8 Conclusion

Contents

Updating a running applications without affecting its normal execution is known as Dynamic Software Update (DSU). This thesis focuses in applying DSU tools to production and to live programming environments, particularly for reflective Object-Oriented languages.

Using DSU solutions in any of the above scenarios presents a number of challenges and requirements. The set of requirements differs when the application is in a production environment or in a development environment.

- Updating production applications requires to minimize the downtime of the execution, to guarantee the stability of the application and to minimize the memory footprint. The patch generation is carefully built by the developer. Implementing the patch requires a profound understanding of the changes and the impact of them. Also, the update of long running applications requires carefully migration of the application objects and the execution of the update only when the application is in a safe state.
- Live programming a stateful application in a development environment requires a deep integration with the Integrated Development Environment (IDE). This integration is required to automatically generate the migration and change logic. In a live programming environment, the patch is generated by the DSU tool. Also, live programming environments require to have a proper isolation of the changes and a seamless integration with existing tools as automatic refactorings.

There exists DSU solutions for each of the above scenarios. However, the existing solutions are designed to resolve the challenges and comply the requirements for only one of the environments. So, we divide the existing solutions in two clear categories: *Production DSU* and *Development DSU*. Analysing these categories we realise that there is not existing solution that addresses the requirements for both scenarios.

We introduce *gDSU*, a novel unifying DSU solution that is applicable in production and live programming scenarios. Our proposed solution has been designed to update long running stateful applications in both scenarios.

gDSU minimizes the manual development and provides a seamless integration with the IDE. We show the applicability of *gDSU* in both scenarios. Also, we present a set of techniques that used by *gDSU* gives it the required performance and minimal memory footprint to be used in production and live programming environment.

8.1 Contributions

This section lists the main contributions of this thesis:

- Main design principles of *gDSU*.
- Implementation techniques to perform it in a practical way.
- Integration with automatic refactorings
- Transactional support for live programming.

Additionally, we published the results of several facets of our work. We list these publications in Appendix A.

8.1.1 gDSU and its techniques

gDSU is the main contribution of this thesis. It is a DSU designed to be applied in production and development environment. Our solution implements safe update point detection using call stack manipulation and a reusable instance migration mechanism to minimize manual intervention in patch generation. Moreover, it also offers updates of core language libraries and the update mechanism itself. This is achieved by the incremental copy of the modified objects and an atomic commit operation. [TPF$^+$eda].

We show that our solution does not affect the global performance of the application and it presents only a run-time penalty during the update window. Our solution is able to apply an update impacting 100,000 instances in 1 second. In this 1 second, only during 250 milliseconds the application is not responsive. The rest of the time the application runs normally while gDSU is looking for the safe update point. The update only requires to copy the elements that are modified.

To achieve a solution that does not affect the global performance of the application and reduce the impact during the update window a set of techniques has been developed. These techniques include an automatic safe update point detection algorithm; an efficient partial copy of the environment; the proposal of reusable instance state migration and validation logic; and a extensible class building process. *gDSU* leverages the existing bulk instance replacement present in Pharo VM.

8.1.2 Atomic Automatic Refactoring

An important activity of software evolution consists in applying refactorings to enhance the quality of the code without changing its behaviour. Having a proper refactoring tool is a must-to in any professional development environment. In addition, live programming allows faster development than the usual edit- compile-debug process. During live programming sessions, the developer can directly manipulate instances and modify the state of the running program.

However, when a complex refactoring is performed, instances may be corrupted (*i.e.,* their state can be lost). For example, when pushing an instance variable to a superclass there is a moment where the superclass does not have yet acquired the new instance variable and the subclass does not have it anymore. It means that the value assigned to this instance variable in existing instances is lost after the refactoring. This problem is not anecdotal since 36% of the refactorings described in Fowler's catalog corrupt instances when used in a live programming context. There is a need to manually migrate, regenerate or reload instances from persistent sources. This manual fix lowers the usefulness of live programming.

In this context of live programming, we propose, AtomicRefactoring [TPF+edb], a new solution based on *gDSU* to preserve the state of the application after performing refactorings. We provide a working extension to the existing refactoring tool allowing application developers to perform complex refactorings preserving the live state of the running program.

This externsion to *gDSU* extends the usability of a DSU solution in live programming environments.

8.1.3 State-Aware Transactional Live Programming

Live programming environments, such as Pharo, allow developers to modify the code and application state while the application is running. This allows a faster development cycle compared with the *edit-compile-debug* process.

Live programming environments implement simple yet powerful mechanisms to migrate the application state after each change. It allows developers to modify both methods and classes. However, modifying classes with existing instances could lead to an inconsistent application state, because *e.g.,* new instance variables are left uninitialized or with obsolete state. As these modifications are applied while the live application is running, a naïve development session may break the application.

This requires special care from the developer (*i.e.,* staging, sequencing, and doing intermediate changes) to keep the coherence of application state. We propose a novel tool (PTm) that allows the developer to scope her changes

isolating them from the running application [TPF+18b]. For this, PTm extends *gDSU* to create an alternative environment with all the classes, methods and instances that are modified. The developer uses this environment to execute her code isolated from the running application, to validate it without affecting the running environment. Finally, the developer decides to safely discard her changes or to apply them atomically in the running application.

This externsion to *gDSU* extends the usability of a DSU solution in live programming environments.

8.2 Future Work

The use of DSU solution in both live programming environments and production environment present future work to increase its usability during the development and maintenance processes. Having a transparent DSU usage opens several directions for future work that we consider for exploration.

8.2.1 Distributed DSU

Having a DSU that is usable in production scenarios opens the door to the research of how to apply safe updates in distributed system. Our proposed safe update point algorithm could be used to detect local safe points and collaborate to detect the global safe update point. Moreover, having an algorithm that does not impact the execution of the application while searching the safe update point could lead to a way of having a global safe update point detection that does not affect the execution of the whole cluster.

8.2.2 Isolation and Virtualization

Having isolated environments inside the original environment could open the door to efficient virtualization of environments. This feature could allow to efficiently isolate different versions of the application and different applications.

8.2.3 Analysis of Changes

The analysis of changes, as our solution uses when generating the patch and analysing the conflicts, opens the door to analysis of changes and impact of these changes. This analysis could be extended to detect the need of adding test scenarios that cover the changes performed. Also, this information could be integrated with different visualizations to increase the information presented in a review of changes.

8.2.4 Language Evolution

Having the ability of modifying core language libraries opens the door to new ways of evolving a dynamic reflective language. Using it could allow the developer to safely modify the core language and test its changes in a safe environment. Moreover, this ability could lead to easy development of custom languages and extensions leading to a framework of language development and experimentation.

8.2.5 Development Experience

Having support for transactional modifications in a live programming environment opens the door to research in changes in the way the developer interacts with the IDE. It also opens the research of new ways of safely inspecting, modifying and changing the live environment.

Bibliography

[AS14] Sorin Adam and Ulrik Pagh Schultz. Towards interactive, incremental programming of ros nodes. *arXiv preprint arXiv:1412.4714*, 2014. (Cited on page 1.)

[ato] Atom. https://atom.io/. (Cited on page 26.)

[BCD+14] Jérémy Buisson, Everton Calvacante, Fabien Dagnat, Elena Leroux, and Sébastien Martinez. Coqcots & pycots: Non-stopping components for safe dynamic reconfiguration. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 85–90, New York, NY, USA, 2014. ACM. (Cited on page 27.)

[BDLDP+15] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015. (Cited on pages 3 and 71.)

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. (Cited on pages 59, 72, and 91.)

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC'06)*, volume 4406 of *LNCS*, pages 66–90. Springer, August 2007. (Cited on page 57.)

[BFdH+13] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. In *ACM SIGPLAN Notices*, volume 48, pages 95–104. ACM, 2013. (Cited on pages 1 and 2.)

[BFL+14] S Bragagnolo, L Fabresse, J Laval, P Estefó, and N Bouraqadi. Pharos: a ros client for the pharo language, 2014. (Cited on page 1.)

[Bra07] Gilad Bracha. On the interaction of method lookup and scope with inheritance and nesting. In *3rd ECOOP Workshop on Dynamic Languages and Applications*, 2007. (Cited on page 72.)

[Bra10] Gilad Bracha. Newspeak programming language draft specification version 0.06, 2010. (Cited on page 72.)

[CF17] Miguel Campusano and Johan Fabry. Live robot programming: The language, its implementation, and robot api independence. *Science of Computer Programming*, 133:1–19, 2017. (Cited on page 1.)

[chr] Chrome dev tools. https://developer.chrome.com/devtools. (Cited on pages 2 and 26.)

[CJ16] Walter Cazzola and Mehdi Jalili. Dodging unsafe update points in java dynamic software updating systems. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 332–341. IEEE, 2016. (Cited on page 27.)

[CNSG15] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM. (Cited on page 1.)

[CPDD09] Gwenaël Casaccio, Damien Pollet, Marcus Denker, and Stéphane Ducasse. Object spaces for safe image surgery. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'09)*, pages 77–81, New York, USA, 2009. ACM digital library. (Cited on pages 29 and 92.)

[Dav06] Flanagan David. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., fifth edition, 2006. (Cited on page 1.)

[DCD13] Martín Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. In *IWST'13: International Workshop on Smalltalk Technologies 2013*, 2013. (Cited on page 36.)

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. (Cited on page 1.)

[DGL+07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007. (Cited on pages 28 and 92.)

[DHL96] C Dony, M Huchard, and T Libourel. Automatic hierarchies reorganization: an algorithm and case studies with overload-

ing. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, S*, pages 151–176, 1996. (Cited on pages 3 and 71.)

[DJ05] Daniel Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, September 2005. (Cited on pages 3 and 71.)

[DRG+05] Serge Demeyer, Filip Van Rysselberghe, Tudor Gîrba, Jacek Ratzinger, Radu Marinescu, Tom Mens, Bart Du Bois, Dirk Janssens, Stéphane Ducasse, Michele Lanza, Matthias Rieger, Harald Gall, Michel Wermelinger, and Mohammad El-Ramly. The LAN-simulation: A research and teaching example for refactoring. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 123–131, Los Alamitos CA, 2005. IEEE Computer Society Press. (Cited on page 1.)

[fir] Firebug. https://addons.mozilla.org/en-US/firefox/addon/firebug/. (Cited on pages 2 and 26.)

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. (Cited on pages 3, 71, 75, and 135.)

[GJK+12] Allan Raundahl Gregersen, Bo Nørregaard Jørgensen, Kai Koskimies, et al. Javeleon: An integrated platform for dynamic software updating and its application in self-* systems. In *Engineering and Technology (S-CET), 2012 Spring Congress on*, pages 1–9. IEEE, 2012. (Cited on page 22.)

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. (Cited on pages 1 and 24.)

[gra] Grasp. http://www.graspjs.com/. (Cited on page 26.)

[Han03] Christopher Michael Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003. (Cited on page 1.)

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, nov 2005. (Cited on pages 2 and 14.)

[HN12] Christopher M Hayden and Iulian Neamtiu. Report on the third workshop on hot topics in software upgrades

(hotswup'11). *ACM SIGOPS Operating Systems Review*, 46(1):93–99, 2012. (Cited on page 13.)

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. (Cited on pages 3 and 24.)

[KLT03] Jussi Koskinen, Henna Lahtonen, and Tero Tilus. Software maintenance cost estimation and modernization support. In *ELTIS-project*. University of Jyväskylä, 2003. (Cited on page 1.)

[KM85] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985. (Cited on page 28.)

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *Proceedings of ACM conference on Lisp and Functional Programming*, pages 99–105, Nice, 1990. (Cited on page 24.)

[KZN12] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012. (Cited on pages 3, 71, and 77.)

[KZN14] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014. (Cited on pages 3 and 71.)

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985. (Cited on page 1.)

[LH12] Jens Lincke and Robert Hirschfeld. Scoping changes in self-supporting development environments using context-oriented programming. In *Proceedings of the International Workshop on Context-Oriented Programming*, page 2. ACM, 2012. (Cited on pages 28 and 92.)

[Lim14] Jason Lim. Live programming for robotic fabrication. *Journal of Professional Communication*, 3(2), 2014. (Cited on page 1.)

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987. (Cited on page 3.)

[MB09] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In

USENIX Annual Technical Conference, volume 2009, 2009. (Cited on page 27.)

[MB15] Eliot Miranda and Clément Béra. A partial read barrier for efficient support of live object-oriented programming. In *International Symposium on Memory Management (ISMM '15)*, pages 93–104, Portland, United States, June 2015. (Cited on pages 54 and 64.)

[McD07] Sean McDirmid. Living it up with a live programming language. *SIGPLAN Not.*, 42(10):623–638, October 2007. (Cited on page 2.)

[MDB15] Sébastien Martinez, Fabien DAGNAT, and Jérémy Buisson. Pymoult : On-Line Updates for Python Programs. In *ICSEA 2015 : 10th International Conference on Software Engineering Advances*, pages 80 – 85, Barcelone, Spain, nov 2015. (Cited on pages 2, 23, and 26.)

[MDC92] Jacques Malenfant, Christophe Dony, and Pierre Cointe. Behavioral Reflection in a prototype-based language. In A. Yonezawa and B. Smith, editors, *Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures*, pages 143–153, Tokyo, November 1992. RISE and IPA(Japan) + ACM SIGPLAN. (Cited on page 3.)

[MHPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. (Cited on page 77.)

[MHSM12] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 265–280, New York, NY, USA, 2012. ACM. (Cited on page 27.)

[MJD96] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection*, pages 1–20, 1996. (Cited on page 3.)

[MME12] Emili Miedes and Francesc D Munoz-Esco1. Dynamic software update. Technical report, Technical Report ITI-SIDI-2012/004, 2012. (Cited on pages 19 and 49.)

[MRH17] Toni Mattis, Patrick Rein, and Robert Hirschfeld. Edit transactions: Dynamically scoped change sets for controlled updates

in live programming. *The Art, Science, and Engineering of Programming*, 1, 2017. (Cited on pages 28 and 92.)

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transaction on Software Engineering*, 30(2):126–139, 2004. (Cited on pages 3, 71, and 72.)

[NCV+13] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming*, pages 552–576, 2013. (Cited on page 77.)

[NH09] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM. (Cited on pages 5 and 46.)

[nod] Nodemon. https://github.com/remy/nodemon. (Cited on pages 2 and 26.)

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on page 28.)

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658. IEEE, 2002. (Cited on pages 2 and 20.)

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: architecture for component trading and dynamic updating. In *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159)*, pages 43–51, May 1998. (Cited on page 28.)

[PC11] Luıs Pina and Joao Cachopo. Dust'm-dynamic upgrades using software transactional memory. 2011. (Cited on page 21.)

[PDF+14] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Science of Computer Programming*, 2014. (Cited on page 72.)

[PDF+15] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Max Mattone. Virtualization support for dy-

namic core library update. In *Onward! 2015*, 2015. (Cited on pages 2, 4, 5, 24, 72, 91, and 92.)

[PDFB15] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. A bootstrapping infrastructure to build and extend pharo-like languages. In *Onward! 2015*, 2015. (Cited on pages 4 and 72.)

[PH13] Luis Pina and Michael Hicks. Rubah: Efficient, general-purpose dynamic software updating for java. In *HotSWUp*, 2013. (Cited on pages 2, 23, 49, and 54.)

[PH16] Luís Pina and Michael Hicks. Tedsuto: A general framework for testing dynamic software updates. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 278–287. IEEE, 2016. (Cited on page 28.)

[PKC+13] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor-flexible runtime updates of java applications. *Software: Practice and Experience*, 43(2):153–185, 2013. (Cited on pages 2, 23, and 26.)

[PS87] D. Jason Penney and Jacob Stein. Class modification in the gemstone object-oriented DBMS. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 111–117, December 1987. (Cited on pages 27 and 92.)

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997. (Cited on pages 3, 26, 71, and 82.)

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996. (Cited on pages 26, 72, and 80.)

[RGN+12] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 2012. (Cited on page 57.)

[Riv96a] Fred Rivard. Pour un lien d'instanciation dynamique dans les langages à classes. In *JFLA96*. INRIA — collection didactique, January 1996. (Cited on page 4.)

[Riv96b] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996. (Cited on page 24.)

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999. (Cited on page 3.)

[SAM13] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013. (Cited on page 19.)

[San78] Erik Sandewall. Programming in an interactive environment: The "lisp" experience. *ACM Comput. Surv.*, 10(1):35–71, March 1978. (Cited on pages 1, 24, and 91.)

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, June 2009. (Cited on pages 21 and 77.)

[Shn83] B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, August 1983. (Cited on page 2.)

[Ste90] Guy L. Steele. *Common Lisp The Language*. Digital Press, second edition, 1990. (Cited on pages 1 and 24.)

[Str] The strongtalk type system for smalltalk. http://bracha.org/nwst.html. (Cited on page 72.)

[Tan90] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127 – 139, 1990. (Cited on page 2.)

[TB01] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001. (Cited on page 72.)

[TPF+16] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Instance migration in dynamic software update. In *Workshop on Meta-Programming Techniques and Reflection 2016*, 2016. (Cited on pages 4, 93, and 94.)

[TPF+18a] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Implementing Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM. In *SAC 2018: Symposium on Applied Computing*, Pau, France, April 2018. (Cited on page 57.)

[TPF+18b] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Ptm: State-aware transactional live programming. In *IWST'18: International Workshop on Smalltalk Technologies*, Cagliari, Italy, September 2018. (Cited on pages 8 and 114.)

[TPF+eda] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Atomic dynamic software update for live programming environments. *Journal of Object Technology*, 2017 submitted. (Cited on pages 7 and 112.)

[TPF+edb] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Preserving instance state during refactorings in live environments. *Future Generation Computer Systems*, 2017 submitted. (Cited on pages 8 and 113.)

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987. (Cited on page 72.)

[VCN+12] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press. (Cited on page 77.)

[vis] Microsoft visual studio. https://www.visualstudio.com/. (Cited on page 26.)

[Web] Jetbrains webstorm. https://www.jetbrains.com/webstorm/. (Cited on page 26.)

[WLN13] Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental dynamic updates with first-class contexts. *Journal of Object Technology*, 12(3):1:1–27, August 2013. (Cited on pages 27, 28, and 92.)

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM. (Cited on pages 2, 21, and 77.)

[XS06] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported-an eclipse case study. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 458–468. IEEE, 2006. (Cited on pages 3 and 71.)

[Zer12] ZeroTurnAround. What developers want: The end of application redeployes. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012. (Cited on pages 2, 22, and 77.)

Published Papers

A.1 Journals

Dynamic Software Update from Development to Production
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse. Accepted, The Journal of Object Technology, Special Issue META'16, 2018, Impact Factor: 0.58

(Under Revision) *Preserving Instance State during Refactorings in Live Environments*
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse. Future Generation Computer Systems, Impact Factor: 4.639

A.2 Conferences

Implementing Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse. 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018), 2018.

Wollok: Language + IDE for a gentle and industry-aware introduction to OOP
Nicolás Passerini, Carlos Lombardi, Javier Fernandes, Pablo Tesone, Fernando Dodino. Twelfth Latin American Conference on Learning Technologies (LACLO 2017), 2017

A.3 Workshops

PTm: State-aware Transactional Live Programming
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse. In International Workshop on Smalltalk Technologies (IWST 2018), 2018.

Transparent Memory Optimization using Slots
Pablo Tesone, Santiago Bragagnolo, Marcus Denker, Stéphane Ducasse. In International Workshop on Smalltalk Technologies (IWST 2018), 2018.

Instance Migration in Dynamic Software Update
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane

Ducasse. In Workshop on Meta-Programming Techniques and Reflection (META 2016), 2016.

Wollok – Relearning How To Teach Object-Oriented Programming
Javier Fernandes, Nicolás Passerini, Pablo Tesone. In Congreso Nacional Argentino de Ingeniería Informática y Sistemas de Información. (CONAIISI 2015), 2015

An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types
Nicolás Passerini, Pablo Tesone, Stéphane Ducasse. In International Workshop on Smalltalk Technologies (IWST 2014), 2014.

Enhancing Binding-based User Interfaces with Transaction Support
Nicolás Passerini, Javier Fernandes, Ronny De Jesus, Pablo Tesone, Leonardo Gassman. In International Workshops on Foundations of Object-Oriented Languages (FOOL 2013), 2013

Instructions to Reproduce Validations and Benchmarks

To validate the proposed solution we need to use a stateful bench application. We implemented a stateful chat server. This simple application presents all the problems and requirements described in this work. This application allows us to replicate and evaluate the design decisions in our proposed solution.

The bench application is used to validate the following scenarios:

- Updating application code while migrating live instances.
- Updating kernel libraries of the environment.
- Updating the DSU tool itself.
- Benchmarking the DSU implementation.

B.1 Installation

The bench application, tests scripts and more documentation are available in a GitHub repository. This repository is located in https://github.com/tesonep/chatServer.

The first step to install the bench application is to clone the given repository in a local machine. For the results included in this paper the validations have been executed in a machine running OS X 10.12.6 having a 2,6 GHz Intel Core i7 and 8 Gb of 1600 MHz RAM memory. However, the same validations will run in Linux or Windows machines.

The bench application runs in Pharo 6.1[1]. A basic knowledge of Pharo is required to execute the validations. The documentation of Pharo and beginner instructions are also available in the Pharo web site.

A 32-bits image and VM are needed to run the validations. They are available at the download site.

[1] http://pharo.org

On this fresh image the bench application and the DSU tool should be installed. To do so, Listing B.1 should be executed in the image.

```
EpMonitor current disable.
Deprecation showWarning: false.
Deprecation raiseWarning: false.

Metacello new
  baseline: 'ChatServer';
  repository: 'filetree://pathToClonedRepository';
  load.

EpMonitor current enable.
```

Listing B.1: Installing Bench Application

This script will install the bench application and all the required libraries. Including the DSU tool. For the DSU tool it uses a release called *JOTPaperVersion*.

To simplify the installation in the repository we provide a script that download the required elements and install them inside the *build* subdirectory. The file install.sh is the simplified installation script.

B.2 Executing Validations

All the validations have been executed using the same image. For running the image in development mode there is a script in the root of the repository called run.sh. This script runs the image in interactive mode.

Also this script opens a Pharo Playground to execute different statements. The Playground is a REPL to execute Smalltalk code. It is similar to the Scala Worksheet or the old Workspace in other Smalltalk dialects. These are the pieces of code to replicate the experiments.

In the Pharo Playground to evaluate a line, the line should be selected and with the context menu execute *Do it*.

B.2.1 Preparation

To execute the different validations, live instances are needed. Generation of instances is performed by executing listing B.2. This script generates the users and messages instances to execute the validations. The number of instances can be modified to reflect different scenarios.

```
"Generate Instances"
ChatUpdate new generateInstances: 10000.
```

Listing B.2: Generating Bench Instances

The generated instances, and the code of the application is accessible to browse. Listing B.3 presents the code needed to open the instance inspector and the source code browser. Checking the instances is needed to see if the instance migration is correctly performed.

```
"Inspect User Instances"
ChUser allInstances inspect.
"Inspect Message Instances"
ChMessage allSubInstances inspect.

"Browse Model"
'ChatServer−Model' asPackage browse.
```

Listing B.3: Browsing Code and Instances

B.2.2 Running Validations

For all the validations we have to execute the preparation steps.

For Validation 1, listing B.4 shows the code to apply and revert the application update. Once the application is updated (or reverted) the changes are seen in the instance inspectors and in the source code browser.

```
"Update Model"
ChatUpdate new updateV1ToV2.

"Revert Update Model"
ChatUpdate new updateV2ToV1.
```

Listing B.4: Updating the Bench Application

Even more, there are tests in the code base of the bench application to test it.

For Validation 2, listing B.5 shows the code to apply and revert the update on the DSU tool. After doing the update we can execute any of the other validations to see that the DSU tool is operative.

```
"Update DSU"
ChatUpdate new updateAtomicProcess.

"Revert DSU"
ChatUpdate new revertUpdateAtomicProcess.
```

Listing B.5: Updating the DSU itself

For Validation 3, listing B.6 shows the code to apply and revert the update in the OrderedCollection class, and in the class builder. This classes are used by the DSU process and they are part of the kernel of Pharo language.

```
"Update Kernel libraries"
ChatUpdate new updateKernel.

"Revert Kernel Libraries"
ChatUpdate new revertUpdateKernel.
```

Listing B.6: Updating Core Libraries

```
ChatUpdateMeter >> doTest
  ((0 to: 7) collect:[:e | 10 ** e]) collect:[:q |
      Stdio stdout << q asString.
      Stdio stdout << (q -> (ChatUpdateMeter new testWith:q)) asString.
      Stdio stdout crlf ; flush.
  ].

  Smalltalk garbageCollect.

ChatUpdateMeter >> testWith: numberOfInstances
  | user room duration |

  ChUser initialize.
  ChRoom initialize.

  (1 to: 3 do: [:e | Smalltalk garbageCollect ]).

  user := ChUser registerUser: 'username' firstName: 'firstName' lastName: 'lastName'.
  room := ChRoom addRoom: 'roomName'.

  instances := OrderedCollection new: numberOfInstances.
  1 to:numberOfInstances do:[:i |
    instances add:(i % 2  = 0 ifTrue:[
      ChMessage from: user to: room text: 'Generic user message'
    ] ifFalse:[
      ChMessage in: room text: 'Generic info message'.
    ])
  ].

  self assert: instances size = numberOfInstances.
  duration := [ChatUpdate new updateV1ToV2] timeToRun .
  "The garbage collector runs many times to force the instances
  to move to the old space. Doing so we test a scenario that
  is closer to a long running application."
  (1 to: 3 do: [:e | Smalltalk garbageCollect ]).

  "Perform the update"
  ChatUpdate new updateV2ToV1.
  ^ duration.
```

Listing B.7: Memory Consumption Benchmark

To achieve repeatability we have scripted the changes in the methods that

are used for each of the updates. These methods can be easily browsed to see the executed changes. Moreover, these methods can be modified to perform other updates. For example, the 3rd validation updates is implemented in the method updateKernel of the ChatUpdate class.

B.3 Executing Benchmarks

We implemented two different benchmarks, the first one measures the memory consumption of the DSU process and the second one the downtime during an update.

B.3.1 Memory Consumption

To measure memory consumption, we have implemented a benchmark that executes the application update 8 times. Using 1 to 10,000,000 live instances (using 10^n where n is in [0,8]). Listing B.7 shows the code executing during this benchmark. This process outputs the results in the standard output of the terminal. Listing B.8 shows the code to evaluate to run the benchmark.

```
"Executing memory and time benchmark per instance quantity. (Long to execute). It
    outputs in the Standard Output the different test executed, listing number of instances
    and time consumed. From 1 instance to 10.000.000"

ChatUpdateMeter new doTest.
```

Listing B.8: Launching Memory Consumption Benchmark

B.3.2 Server Response Time

The second benchmark is designed to validate the downtime of the application during an update. In this benchmark, the application is running in server mode. It implements a REST server to receive request through HTTP.

The server is launched and stopped from the image. Listing B.9 shows the Smalltalk code to start and stop the server instance.

```
"Starting the HTTP test to run the benchmark with JMeter. The REST server is listening in
    port 1701"
ChatServer uniqueInstance start.

"Stopping server"
ChatServer uniqueInstance stop.
```

Listing B.9: Controlling the Bench Server

Once the server is started, the update process can be requested via REST calls. We implemented a different REST request for each of the updates. Ta-

REST URL	**Action**
http://localhost:1701/updateV1ToV2	Updates from V1 to V2.
http://localhost:1701/updateV2ToV1	Updates from V2 to V1.
http://localhost:1701/updateKernel	Updates the Kernel implementation of OrderedCollection.
http://localhost:1701/revertUpdateKernel	Reverts the Kernel implementation of OrderedCollection.
http://localhost:1701/updateAtomicProcess	Updates the DSU implementation.
http://localhost:1701/revertUpdateAtomicProcess	Reverts the DSU implementation.

Table B.1: Update REST Entry points

ble B.1 shows the REST entry points that can be used to perform different updates while the server is running.

To simulate the load of the server we use a JMeter[2] script. This script is designed to perform 10 concurrent requests during 2 minutes. It generates an average of 700 requests per second. The script is located in the root of the git repository, and it is named chatServer.jmx. We refer to the documentation of JMeter on how to run the given script.

In the benchmark, we execute the JMeter script and after one minute we perform one of the update REST calls. Once the JMeter scripts ends it presents the results of the benchmark.

[2]http://jmeter.apache.org/

Detailed Analysis of Automatic Refactorings

After analysing the impact to live instances of the refactorings described in *Refactoring: Improving the Design of Existing Code* [Fow99], we present here the detailed results of all the refactorings in the book. There are 46 refactorings that does not affect live instances, 6 refactorings with complex corruption, 11 with class corruption, and 9 with internal corruption.

C.1 Refactoring without Corruption

The following refactorings do not have any impact in live instances. The class structures are not affected.

Refactoring	**Page**
Add Parameter	275
Change Reference to Value	183
Consolidate Conditional Expression	240
Consolidate Duplicate Conditional Fragments	243
Convert Procedural Design to Objects	368
Decompose Conditional	238
Encapsulate Collection	208
Encapsulate Downcast	308
Encapsulate Field	206
Extract Interface	341
Extract Method	110
Form Template Method	345
Hide Delegate	157
Hide Method	303
Inline Method	117
Inline Temp	119
Introduce Assertion	267
Introduce Explaining Variable	124
Introduce Foreign Method	162

Refactoring	**Page**
Introduce Parameter Object	295
Move Method	142
Parameterize Method	283
Preserve Whole Object	288
Pull Up Constructor Body	325
Pull Up Method	322
Push Down Method	328
Remove Assignments to Parameters	131
Remove Control Flag	245
Remove Middle Man	160
Remove Parameter	277
Remove Setting Method	300
Rename Method	273
Replace Constructor with Factory Method	304
Replace Error Code with Exception	310
Replace Exception with Test	315
Replace Magic Number with Symbolic Constant	204
Replace Method with Method Object	135
Replace Nested Conditional with Guard Clauses	250
Replace Parameter with Explicit Methods	285
Replace Parameter with Method	292
Replace Record with Data Class	217
Replace Temp with Query	120
Self Encapsulate Field	171
Separate Query from Modifier	279
Split Temporary Variable	128
Substitute Algorithm	139

C.2 Refactoring with Complex Corruption

Refactoring	Page	Explanation
Change Bidirectional Association to Unidirectional	200	One of the sides of the bidirectional association should be dropped. One of the two classes involved in the association will drop one instance variable.
Change Unidirectional Association to Bidirectional	197	The modified instances need the objects referencing to them to construct the bidirectional association. This information is not present in the modified instances.
Change Value to Reference	179	All the client instances of this object should be updated to reference the same object.
Introduce Null Object	260	All the clients using *null* in this field should be updated to use the newly created object. If this object should be shared, it should be done in the migration process.
Move Field	146	The field can come from any class in the system, the original and target classes are not related. The logic to match from the original instances to the target instances should be in the migration process.
Replace Data Value with Object	175	It exposes the same problems of *Extract Class* but the new instances should be shared, it makes more complex the migration process.
Replace Type Code with Class	218	It exposes the same problems of *Extract Class* but the new instances should be shared, it makes more complex the migration process.

C.3 Refactoring with Class Corruption

Refactoring	Page	Explanation
Collapse Hierarchy	344	All the instances of the subclass should be migrated to the superclass. As the subclass does not exist any more.
Extract Hierarchy	375	Some of the modified instances should be migrated to the new subclasses. The migration process should manage the determination of which class to instantiate in each case.
Extract Subclass	330	Some instances should be migrated to the new subclass. Also, the structure of the main class is changed.
Extract Superclass	336	A new superclass is extracted from the common part in two classes. The live instances need a migration if the class structure changed.
Introduce Local Extension	164	Some instances should be migrated to the new class. Establishing which instances to migrate is a responsibility of the migration process.
Replace Conditional with Polymorphism	255	A new set of subclasses are created, the live instances should be migrated to these subclasses according to the values of the original instance variables. Also, the instance variables of the subclasses might be renamed.

Refactoring	**Page**	**Explanation**
Replace Delegation with Inheritance	355	A pair of collaborating objects is integrated into the same hierarchy. Making the client of the delegation a subclass of the delegate. The instances of the client should be migrated to the new subclass, and all the instance variables of the delegate should be migrated to the client.
Replace Inheritance with Delegation	352	As a subclassification is replaced with a delegation, all the internal state of the single instance should be migrated to the new collaboration. Also, the delegate object should be created.
Replace a Subclass with Fields	232	All the instances of the subclass should be migrated to the superclass. Preserving the subclass state and adding the needed fields to distinguish from the superclass instances.
Replace Type Code with Subclasses	223	The instances should be migrated to new subclasses depending on the value of the type code.
Tease Apart Inheritance	362	As the hierarchy is split, the live instances should be split in the same way, putting the original instance state to the corresponding instances.

C.4 Refactoring with Internal Corruption

Refactoring	Page	Explanation
Duplicate Observed Data	189	As *Extract Class*, with the addition that the extracted has a reference to the original instance it was extracted from.
Extract Class	149	The live instances of the original class should be split in the new version of the mother instances and the newly created child instance.
Inline Class	154	The internal state of the child object should be migrated to the mother instance.
Pull Up Field	320	The structure of the class has changed, all the fields should be migrated to their new position in the object.
Push Down Field	329	The structure of the class has changed, all the fields should be migrated to their new position in the object.
Replace Array with object	186	The same problematic of *Extract Class*.
Replace Type Code with State/Strategy	227	The same problematic of *Extract Class*, only having more possible subclasses.
Separate Domain From Presentation	370	The same problematic of *Extract Class*.

Dynamic Software Update for Production and Live Programming Environments

Pablo Tesone

Abstract: Updating applications during their execution is used both in production to minimize application downtime and in integrated development environments to provide live programming support. Nevertheless, these two scenarios present different challenges making Dynamic Software Update (DSU) solutions to be specifically designed for only one of these use cases. For example, DSUs for live programming typically do not implement safe point detection or instance migration, while production DSUs require manual generation of patches and lack IDE integration. These solutions also have a limited ability to update themselves or the language core libraries and some of them present execution penalties outside the update window.

In this PhD, we propose a unified DSU named *gDSU* for both live programming and production environments. *gDSU* provides safe update point detection using call stack manipulation and a reusable instance migration mechanism to minimize manual intervention in patch generation. It also supports updating the core language libraries as well as the update mechanism itself thanks to its incremental copy of the modified objects and its atomic commit operation.

gDSU does not affect the global performance of the application and it presents only a run-time penalty during the update window. For example, *gDSU* is able to apply an update impacting 100,000 instances in 1 second making the application not responsive for only 250 milliseconds. The rest of the time the application runs normally while *gDSU* is looking for a safe update point during which modified elements will be copied.

We also present extensions of *gDSU* to support transactional live programming and atomic automatic refactorings which increase the usability of live programming environments.

Keywords: dynamic software update, live programming, long running applications, transactional modifications, automatic refactorings.

Mise à jour Dynamique pour Environnements de Production et Programmation Interactive

Pablo Tesone

Résumé: Mettre à jour des applications durant leur exécution est utilisé aussi bien en production pour réduire les temps d'arrêt des applications que dans des environnements de développement interactifs (IDE pour *live programming*). Toutefois, ces deux scénarios présentent des défis différents qui font que les solutions de mise à jour dynamique (DSU pour *Dynamic Software Updating*) existantes sont souvent spécifiques à l'un des deux. Par exemple, les DSUs pour la programmation interactive ne supportent généralement pas la détection automatique de points sûrs de mise à jour ni la migration d'instances, alors que les DSUs pour la production nécessitent une génération manuelle de l'ensemble des modifications et manquent d'intégration avec l'IDE. Les solutions existantes ont également une capacité limitée à se mettre à jour elles-mêmes ou à mettre à jour les bibliothèques de base du langage; et certaines d'entre elles introduisent même une dégradation des performances d'exécution en dehors du processus de mise à jour.

Dans cette thèse, nous proposons un DSU (nommé *gDSU*) unifié qui fonctionne à la fois pour la programmation interactive et les environnements de production. *gDSU* permet la détection automatique des points sûrs de mise à jour en analysant et manipulant la pile d'exécution, et offre un mécanisme réutilisable de migration d'instances afin de minimiser les interventions manuelles lors de l'application d'une migration. *gDSU* supporte également la mise à jour des bibliothèques du noyau du langage et du mécanisme de mise à jour lui-même. Ceci est réalisé par une copie incrémentale des objets à modifier et une application atomique de ces modifications.

gDSU n'affecte pas les performances globales de l'application et ne présente qu'une pénalité d'exécution lors processus de mise à jour. Par exemple, *gDSU* est capable d'appliquer une mise à jour sur 100 000 instances en 1 seconde. Durant cette seconde, l'application ne répond pas pendant 250 milli-secondes seulement. Le reste du temps, l'application s'exécute normalement pendant que *gDSU* recherche un point sûr de mise à jour qui consiste alors uniquement à copier les éléments modifiés.

Nous présentons également deux extensions de *gDSU* permettant un meilleur support du développement interactif dans les IDEs : la programmation interactive transactionnelle et l'application atomique de reusinages (*refactorings*).

Mots clés: mise à jour dynamique, programmation interactive, applications de longue durée, modifications transactionnelles, réusinage de code